# Semantics-based parallel cost models and their use in provably efficient implementations

John Greiner

April 26, 1997

CMU-CS-97-113

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

**Thesis committee:**
Guy Blelloch, Chair
Robert Harper
Gary Miller
Guy Steele, Jr., Sun Microsystems

This research was sponsored in part by the Wright Laboratory, Aeronautical Systmes Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grand number F33615-93-1-1330 and contract number F19628-91-C-0168. It was also supported in part by an NSF Young Investigator Award and by Finmeccanica.

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Wright Laboratory or the U.S. Government.

**Keywords:** Functional languages, parallel algorithms, lambda calculus, models of computation, computer architecture

# Abstract

Understanding the performance issues of modern programming language execution can be difficult. These languages have abstract features, such as higher-order functions, laziness, and objects, that ease programming, but which make their mapping to the underlying machine more difficult. Understanding parallel languages is further complicated by the need to describe what computations are performed in parallel and how they are affected by communication and latency in the machine. This lack of understanding can obscure even the asymptotic performance of a program and can also hide performance bugs in the language implementation.

The dissertation introduces a framework of provably efficient implementations in which performance issues of a language can be defined and analyzed. We define several language models, each consisting of an operational semantics augmented with the costs of execution. In particular, the dissertation examines three functional languages based on fork-and-join parallelism, speculative parallelism, and data-parallelism, and it examines their time and space costs. We then define implementations of each language model onto several common machine models, prove these implementations correct, and derive their costs.

Each of these implementations uses an intermediate model based on an abstract machine to stage the overall implementation. The abstract machine executes a series of steps transforming a stack of active states and store into new states and store. The dissertation proves the efficiency of the implementation by relating the steps to the parallel traversal of a computation graph defined in the augmented operational semantics.

Provably efficient implementations are useful for programmers, language implementors, and language designers. For example, they provide a formal definition of language and implementation costs for program analysis, compiler specification, and language comparisons. The dissertation describes performance problems in existing implementations of Id and NESL and gives provably more efficient alternatives for each. It also compares the example language models, first using several specific algorithms, and also in more generality, for example, quantifying the speedup obtainable in the data-parallel language relative to the fork-and-join language.

# Acknowledgements

I'd like to acknowledge and thank all the people that helped make this dissertation. In particular, kudos to my advisors, Guy Blelloch and Bob Harper, for goading and guiding me, and to the rest of my thesis committee, Gary Miller and Guy Steele, for their patience. Pseudo-officemates Mark and Mark helped provide ideas and feedback on just about anything.

Without my friends I'd never have stuck around so long, so special thanks to officemates Dave, Sing Bing, and Chris; my previously mentioned pseudo-officemates, who were probably sick of me always dropping by; the "Rubber Ducky" classmates; my roommates; and the whole volleyball gang.

Boos and hisses to Usenet and the Web for forcing me to spend so much time away from real work. It couldn't have been my fault, could it?

# Contents

# List of Figures

# Part I

# Introduction

# Chapter 1

# Introduction

The primary goal of this dissertation is to understand the performance issues of modern programming languages. To achieve part of this overall goal, we show a framework for defining and analyzing the asymptotic performance issues of programming models. Using this framework, we give provably efficient implementations of several languages, each on several machines. In particular, for each language we

1. define an abstract notion of computation in the language, including not only what result is computed for a program, but also an abstract model of *how* it is computed;
2. use standard definitions of machines and their notions of computation; and
3. provide implementations of the language on these machines, with proofs that the implementation preserves correctness and mappings between a program's costs of computation in the language and in the machines.

We discuss three parallel functional languages, each with a different model of parallelism, and their time and space costs for program execution. We implement each of these on three standard parallel machines. However, the framework generalizes beyond these specifics to other languages, cost models, and machines.

## 1.1 Background and Problems

A *semantics* defines a programming language. A traditional *extensional* semantics defines a program's results, including any input/output behavior, and its termination properties. Extensional semantics are well-understood for a wide variety of languages. On the other hand, an *intensional* semantics defines an abstract model of how a computation is performed, such as how long a computation takes or the resources needed during a computation. An intensional semantics that tracks run-time cost information is called a *profiling* semantics [108, 110].

A semantics can also be considered a simple abstract implementation of a language. This is especially true for *operational* styles of semantics, which are of primary interest here. These simple implementations do not necessarily embody the intensional properties expected of a realistic implementation.

Implicitly or explicitly, some intensional properties are considered to be part of the language itself and not to depend on the implementation. Many implicit implementation requirements are considered common sense and obvious, *e.g.*, that adding two integers should take constant time. This example assumption is reasonable for fixed-precision arithmetic, but not for the arbitrary-precision arithmetic available in some languages, such as Scheme [27] and Mathematica. Since many intensional properties really are "obvious" in most commonly used languages, such as C or Fortran, explicitly defining these properties is not considered a priority. But not all intensional properties are "obvious", especially in modern programming languages that are more abstract than C, Fortran, *etc.*

Many constraints on languages can be considered required "optimizations". The best example of this is Scheme's explicit requirement for tail recursion [27], *i.e.*, *tail calls* in a function are implemented with a jump rather than a function call. The execution of a tail-recursive function reuses the current stack frame so that a sequence of tail calls requires only one stack frame. Thus, the requirement affects the space used by a program in an asymptotically significant way. This is a common optimization in functional languages, that is, by definition, required in Scheme.

Thus, some implementation decisions may be considered essential to a given language, especially if they significantly affect the run-time costs of the language. The following examples hint at the range of languages and properties that are of interest:

- In languages with more complicated numbers than fixed-size integers or floating point numbers, especially those with arbitrary-precision arithmetic, how long do basic arithmetic functions take?
- In parallel languages, what is or can be executed in parallel, and how many processors can be kept busy?
- In parallel languages, how does the space usage depend on the number of processors available? (Using more processors generally means that at any given moment, more control information is used and more live data is accessible.)

## 1.2 Provably Efficient Implementations

This dissertation introduces *provably efficient implementations* to specify the intensional properties and implementations of languages and prove efficiency results about these implementations. The profiling semantics is an abstract specification of the intensional costs. Cost mappings then relate these to the costs on more concrete machine models. We must prove the use of the cost mappings is feasible by providing an abstract implementation that obtains the desired bounds, as in Figure 1.1. While we are primarily concerned with intentional properties, we also specify and prove results about the models' extensional properties. However, the extensional results shown in this dissertation are not surprising.

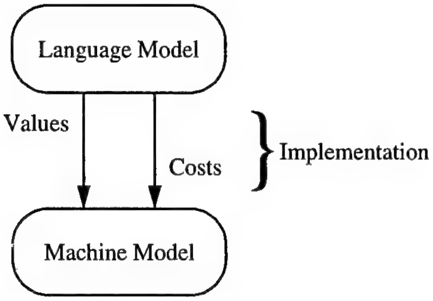


Figure 1.1: The implementation maps values and costs of the profiling semantics to those of the machine. Its effects on costs are summarized by cost mappings.

### 1.2.1 Uses of provably efficient implementations

To describe some uses of a provably efficient implementation and its components, we consider three different perspectives: the language designer, the language implementor and tool developer, and the language user (*i.e.*, programmer).

The designer creates the profiling semantics and proves the cost mappings. These intensional formalisms allow the designer to specify run-time properties such as Scheme's tail recursion in a formal and well-defined way.

The implementor uses the cost mapping and its feasibility proof as an abstract specification of the compiler. He can use the cost mapping to verify an implementation's compliance with the specification. Furthermore, he uses the profiling semantics as an abstract definition of costs for analyses within the compiler, profilers, and automatic complexity analyzers.

The programmer uses the profiling semantics as the formal definition of understanding a program's behavior, results, and execution costs. The programmer uses the cost mapping to summarize whatever he is expected to know about the compiler. He can use the cost mapping to compare the profiling semantics' abstract notion of costs to what happens on various machines. Thus, he would perform a single cost analysis in the abstract language model, even if the program is targeted for multiple machine models. In particular, this dissertation is concerned with using the profiling semantics to analyze asymptotic performance and compare algorithms, such as whether to use quicksort or insertion sort.

### 1.2.2 Limiting our scope

To limit the scope of the thesis, we restrict our attention to parallel models based on functional languages and their asymptotic costs.

- We use purely functional languages because of their simple semantics, which can be described with relatively few and simple rules.

- We use parallel models because the run-time costs are much less understood than those of serial models. Frequently, it is unclear which subcomputations are executed in serial or in parallel, as this can be dependent on how long certain subcomputations take, how many processors the machine has, how long communication delays are in the machine, *etc.* Furthermore, unlike serial machines, which are almost all relatively similar, parallel machine architectures can be radically different from each other.

  Here, by "parallel" we mean that programs are to be executed on multiprocessor ("parallel") machines. The languages we use are not semantically parallel, but sequential, *i.e.*, they do not include constructs such as *parallel-or*[1]. Thus, these languages are deterministic and not concurrent.

  Side-effect-free applicative languages are a natural candidate for modeling parallelism since it is always safe to evaluate subexpressions in parallel in these languages [39, 40].

- We examine asymptotic costs because this allows us to simplify many issues by ignoring constant factors. Even the asymptotic cost bounds of languages and their implementations are not well understood, and many implementation decisions affect the run-time costs in an asymptotically significant manner. For example,

  - tail recursion asymptotically affects stack space;
  - an optimization in some versions of Standard ML of New Jersey to share the space for function environments keeps data accessible for too long, increasing space usage asymptotically[5]; and
  - the implementations of some parallel languages needlessly serialize the synchronization of threads, asymptotically reducing the parallelization of some programs (*cf.* Chapter 8).

While we restrict ourselves to such languages in this dissertation, the framework of provably efficient implementations is applicable to any language and its implementation. Also, while we could include more details to account for constant cost factors, that would obscure the ideas of primary interest here.

### 1.2.3 Models of parallelism

This dissertation describes three basic models of parallelism. Each is based on the *pure* (*i.e.*, no side-effects) *$\lambda$-calculus*, where an expression $e$ is one of the following: a constant $c$, a variable $x$, a function of one argument $\lambda x.e'$, or application $e_1\ e_2$ of function $e_1$ and argument $e_2$. Additional constructs such as data structures, conditionals, and recursion can be included easily, or they may be simulated with the core using standard techniques. The $\lambda$-calculus, though powerful, offers a simplicity that makes it easy to reason about. It is

[1] Parallel-or takes two argument expressions and returns **true** if either argument returns **true**, even if the other argument never terminates. Implementing this requires some form of concurrency.

the direct ancestor of functional languages such as Scheme, ML, and Haskell, and it is also commonly used as a meta-language for defining the semantics of imperative, object-oriented, logic-based, and other languages.

The models of parallelism we examine are as follows:

- Fork-and-join parallelism allows a bounded number of threads to be forked (spawned) and later joined (synchronized) at a specified point in the control flow. These threads can, in turn, fork and join additional threads in a strictly nested fashion, and each thread can be evaluated in parallel. The *Parallel Applicative $\lambda$-calculus* (PAL) allows two threads to be forked, and later joined, by an application expression: one each for $e_1$ and $e_2$. The PAL uses call-by-value application, so $e_2$ is fully evaluated before being applied to the result of evaluating $e_1$.

  We also show how using an extended syntax instead results in a model (PAL') that is equivalent up to constant factors of their asymptotic costs.

  All data structures are pointer-based, so all data must be stored in tree- or list-like structures. In many programs, a balanced binary tree leads to the most efficient algorithm.

- Speculative parallelism (or *call-by-speculation* [53]), as used here, also allows a bounded number of threads to be spawned at once. However, it synchronizes only as necessary for data dependencies, *i.e.*, arguments are evaluated in parallel with function application and evaluation of its body. This allows "pipelined" and "producer-consumer" parallelism to be expressed.

  Since synchronization is more relaxed than in fork-and-join parallelism, this can allow faster programs, but since synchronization is data-dependent, it is more difficult to formally define and implement. This style of parallelism is closely related to the *futures* of Multilisp [50] (also known as *promises* [40])—Multilisp applies an application's function to futures which represent the arguments and which eventually receive the arguments' final values. Speculative parallelism also forms the core of languages such as Id and pH [87, 2, 88]. The *Parallel Speculative $\lambda$-calculus* (PSL) allows two threads to be spawned by an application expression, as in the PAL. Synchronization occurs only when looking up a variable's value.

  The basic form of *full* speculation (PSLf) eventually evaluates all spawned threads, and thus requires the same amount of computation as the PAL model. The alternative of *partial* speculation (PSLp) allows *irrelevant* computations to be aborted and discarded, potentially reducing the amount of computation. Partial speculation is a family of models differing in how we detect and abort irrelevant computations, generally prioritizing computations so as to reduce the amount of computation spent on irrelevant ones. Note that call-by-need is one extreme of partial speculation, where computations are prioritized to ensure that we do not evaluate any irrelevant computations.

- Data-parallelism allows the forking and joining of an unbounded number of copies of an expression, where we give each copy a different piece of data. The NESL model uses call-by-value application, although unlike the PAL model, applications are not parallelized. Instead, we introduce sequences as a datatype and an expression $\{e'' : x \textbf{ in } e'\}$ which evaluates $e''$ in parallel for each binding of $x$ to a value in the sequence resulting from $e'$. This forms the core of languages such as NESL [14], HPF, and C* [107]. NESL provides a very flexible model of data-parallelism, where $e''$ may be any general expression. In particular we allow nested data-parallelism, *i.e.*, forked threads can fork additional threads (as in NESL, but not HPF and C*).

  We show that in quicksort, for example, NESL allows more efficient data access and thus more efficient algorithms than the PAL or PSL models.

We do not use any call-by-name or call-by-need (lazy) languages, because they inherently do not offer significant parallelism [64, 121]. In fact, parallel graph reduction, a form of partial speculation, is generally offered as a compromise of laziness to obtain parallelism.

### 1.2.4 Costs of parallelism

This dissertation describes and proves results about the time costs of each of the PAL, PSL, and NESL models and the space costs of the PAL and NESL models. This section outlines how we define and use the abstract costs of time and space. The following two sections then outline how we incorporate these definitions in the models and relate them to the machine models' costs, respectively.

We are interested in how these costs depend upon the size of the input. Furthermore, since we use parallel machine models, we are also interested in how these costs depend upon the number of machine processors, as discussed when we relate them to the machine model's costs.

#### Computation graphs

*Computation graphs* are directed acyclic graphs, where nodes represent units of computation, and edges represent data and control dependences. Computation graphs provide an intuitive visual summary of computation; formally generalize *work* and *depth*, to be described; and describe the computation scheduling. Figure 1.2 gives an example. Each of our models' profiling semantics defines the computation graph of programs, and Figure 1.3 illustrates the different models of parallelism that they represent.

#### Work and Depth

To describe the time costs, we use two cost measures, work and depth. The work is the number of units of computation executed in an evaluation, independent of whether the computation is performed in serial or parallel. The depth is the one plus the total length of the longest

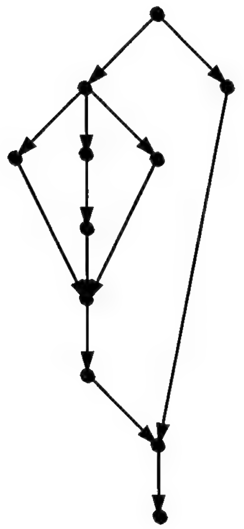


Figure 1.2: Example computation graph. Nodes represent units of computation, and edges represent data and control dependences.

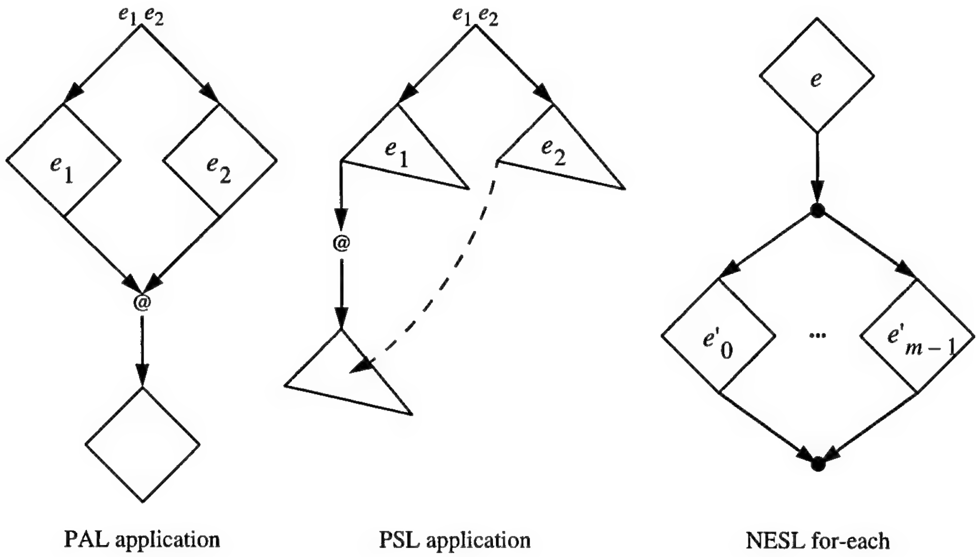


Figure 1.3: Simplified illustration of parallelism in the PAL, PSL, and NESL models, where diamonds and triangles represent subgraphs for subexpression computation. The dashed line represents synchronization for a potential data-dependency. The "@" nodes each represent the application of a function value. Additional details are provided in later chapters.

```
fun quicksort xs =
    if size of xs is less than 2 then xs
    else let pivot = median element of xs
             lts   = elements in xs   less than  pivot
             eqs   = elements in xs   equal to   pivot
             gts   = elements in xs greater than pivot
             s_lts = quicksort lts
             s_gts = quicksort gts
         in append s_lts eqs s_gts
```

Figure 1.4: Quicksort pseudo-code. The three selections of elements less than, equal, and greater than the pivot are each independent and can be parallelized with respect to each other. The two recursive calls can be parallelized. Each of the subtasks of choosing the pivot and selecting elements can be parallelized.

path through the computation, viewing the computation as a graph of data and control dependencies. As an example, refer to Figure 1.2. Its work is the number of nodes ($w = 11$), its depth is the length of the longest path, plus one for the initial node ($d = 8$).

In serial models, having a total order on the control dependencies results in program execution time being equivalent to each of work and depth. In parallel models, a program's execution time is dependent on (at least) its work and depth and the number of processors available. Work and depth are frequently used to describe parallelism, especially in teaching parallel algorithms [63, 59, 17, 15] and implementing various applications [46, 11, 10].

Let's examine the work and depth of quicksort on $m$ data elements. Figure 1.4 gives pseudo-code for a parallel quicksort. First, recall that a serial quicksort algorithm requires $O(m \log m)$ time in the expected case. For any of our language models, the parallel quicksort requires $O(m \log m)$ work (expected), just as for the serial quicksort. Each recursive iteration is dominated by the $O(m)$ work to examine each of the elements, and there are two recursive calls each on half (on average) of the elements, thus the $W(m) = O(m) + 2W(m/2) = O(m \log m)$ (expected) total. Chapter 10 discusses quicksort more formally.

The depth of quicksort depends on the data structure to store the elements. In the PAL model, we can choose between lists and trees as data structures. Using lists, splitting the elements on the pivot and appending the sorted lists each requires linear depth, resulting in a total of $D(m) = O(m) + D(m/2) = O(m)$ depth (expected). Using balanced binary trees, splitting and appending the elements requires logarithmic depth, resulting in a total of $D(m) = O(\log m) + D(m/2) = O(\log^2 m)$ depth (expected), as shown in Corollary 10.1. These same bounds also hold for the PSL model, as the algorithm has no significant amount of pipelining available.

For comparison, it is possible to sort in $O(\log m)$ depth in the PAL model. In the *counting sort*, each element first compares itself to all other elements and counts how many of those are less than itself. Next, each element then places itself in the position indicated by its count. Assuming the elements are kept in a balanced binary tree, this requires $O(\log m)$ depth, an $O(\log m)$ factor less than than quicksort. However, it requires $O(m^2)$ work for the comparisons. When this work and depth are mapped to the costs of the machine, as in Section 1.2.6, we see that this is only efficient for relatively small values of $m$, *i.e.*, small data sets. In general, we want algorithms that are work-efficient and have low computation depth bounds.

An efficient NESL quicksort algorithm uses sequences to store the data. Choosing a pivot then requires constant depth. Furthermore, the selection of elements less, equal, and greater than the pivot then requires three constant-depth uses of for-each. Appending the sorted sequences also requires constant depth. As a result, the algorithm requires only $D(m) = O(1) + D(m/2) = O(\log m)$ depth.

### Space

The profiling semantics defines the space cost for serial evaluation, and the cost mappings relate this to the space required for parallel evaluation, given the number of machine processors. We formally bound the parallel space in terms of the serial space by relating the machine execution to parallel traversals of the profiling semantics' computation graph and then using previous results about space usage of parallel traversals.

In quicksort, each model only requires reachable space linear in the number of data elements. At any time, only a constant number of copies of the original data is live, for a total of linear space. That dominates the polylogarithmic space needed for the recursion stack of any reasonable choice of data structure.

The semantics do not need to model *garbage collection*, the automatic reclamation of unaccessible memory. Instead, we measure the maximum amount of space during the evaluation required for all *reachable*, or *live*, data and any overhead such as a control stack. Our implementations do not include garbage collection either, because that would require extra details obscuring other features. However, Appendix B outlines how we can add garbage collection and how this affects the implementations' cost bounds.

### Other issues

Modeling time and space costs allows us to examine many issues of implementation efficiency. In particular, this dissertation addresses two problems of previous parallel language implementations. One is the time delay incurred by some serial bottlenecks in *speculative* languages. Another is the space overhead when there is "too much" parallelism. *I.e.*, if many more threads are spawned than there are processors, the space for storing these delayed computations may dominate.

### 1.2.5 Formalizing the cost models

We provide a formal cost model to specify the intensional properties (here, the computation graph and maximum reachable space) of a language. This cost model is incorporated into the semantics of the language, augmenting an extensional semantics with cost definitions, resulting in a profiling semantics. It is from this formal definition that we can derive bounds such as described for quicksort.

The profiling semantics by itself does not reflect the implementation costs of the language. Since the implementation costs depend on the underlying machine, and since we want a single profiling semantics for the language, we also need to formally relate the costs of the language model to those incurred in the machine model. This relation reflects the essential details of the implementation, and is described further in the next section. Together, the profiling semantics and this cost mapping provide the essential intensional information about a language.

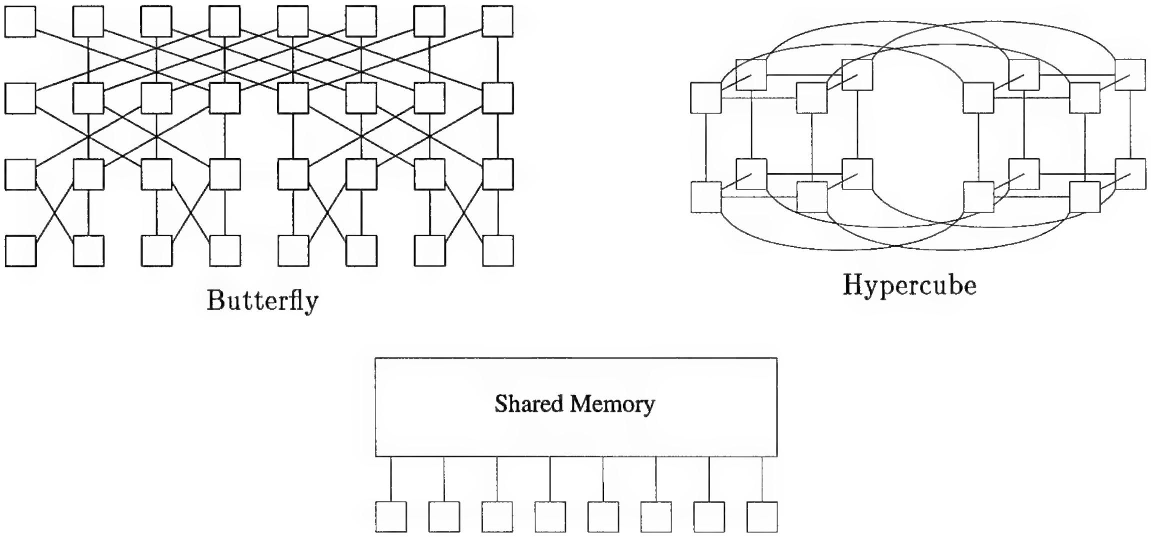
The underlying extensional semantics we use is *operational*, rather than *denotational.* An operational-style semantics defines the result of evaluating an expression to a value in a way that, at least abstractly, matches the evaluation process. A denotational-style semantics defines the "meaning" of an expression as a value in a compositional manner, with no direct appeal to the evaluation process. The operational style matches our needs better, since we are interested in the costs of the evaluation process.

### 1.2.6 Implementations and their cost mappings

This dissertation uses three standard parallel machine models: the butterfly, the hypercube, and the Parallel Random-Access Machine (PRAM) [38]. Each of these uses a collection of processors connected by a different style of communication network, as illustrated in Figure 1.5. The butterfly and hypercube are commonly used in practical networks, while the PRAM is a common abstraction of parallel machine models. We use three kinds of PRAM, differing in how they access memory: the exclusive-read, exclusive-write (EREW); the concurrent-read, exclusive-write (CREW); and the concurrent-read, concurrent-write (CRCW).

In each model we assume memory access and allocation requires constant time. For the butterfly we assume that for $p$ processors we have $p \log_2 p$ switches and $p$ memory banks, and that memory references can be pipelined through the switches. We also assume the butterfly network has simple integer adders in the switches, such that *scan* and *reduce* operations (see Appendix A for definitions) can execute in $O(\log p)$ time. For the hypercube we assume a multiport hypercube in which messages can cross all wires on each time step, and for which there are separate queues for each wire.

The time costs of the implementations are parameterized by the overhead of communication through the communication networks, as modeled by scan and *fetch-and-add* operations (see Appendix A for definition). In the PAL model, the overhead is bounded asymptotically by the time $TS(p)$ for a scan on $p$ processors, whereas in the PSL and NESL model, it is bounded asymptotically by the time $TF(p)$ for the more general fetch-and-add. As a result,



Butterfly

Hypercube

Parallel Random Access Machine (PRAM)

Figure 1.5: Illustrations of the butterfly, hypercube, and Parallel Random Access Machine (PRAM), respectively. The squares represent processors, and the edges represent communication links between them.

| | | $TS(p)$ | $TF(p)$ |
|---|---|---|---|
| Machine Model | Randomized? | Time for scan | Time for fetch-and-add |
| Butterfly | Yes | $O(\log p)$ | $O(\log p)$ |
| Hypercube | Yes | $O(\log p)$ | $O(\log p)$ |
| EREW PRAM | Yes | $O((\log p)^{3/2}/\sqrt{\log \log p})$ | $O((\log p)^{3/2}/\sqrt{\log \log p})$ |
| CREW PRAM | Yes | $O(\log p \log \log p)$ | $O(\log p \log \log p)$ |
| CRCW PRAM | No | $O(\log p/\log \log p)$ | $O(\log p \log \log p)$ |
| CRCW PRAM | Yes | $O(\log p/\log \log p)$ | $O(\log p/\log \log p)$ |

| Language | Time | Space |
|---|---|---|
| PAL | $O(w/p + d \cdot TS(p))$ | $O(s + d \cdot p \cdot TS(p))$ |
| PSLf | $O(w/p + d \cdot TF(p))$ | no bounds shown |
| NESL | $O(w/p + d \cdot TF(p))$ | $O(s + d \cdot p \cdot TF(p))$ |

Figure 1.6: Summary of cost mappings of three language models on several parallel machine models. These bounds are parameterized by the time $TS(p)$ or $TF(p)$ for a scan or fetch-and-add operation, respectively, on a $p$-processor machine. Tighter bounds are shown for some of these machine models.

$TS(p)$ and $TF(p)$ bound the latency through the network, and thus the amount of multithreading needed to hide latency. Figure 1.6 summarizes some of the cost mappings obtained in the various models. Note that to provide an efficient fetch-and-add operation, we generally consider only randomized machine models, so these bounds hold with high probability.[2]

We can plug the work and depth bounds of our quicksort example into these mappings. For example, quicksort requires $O(m \log m)$ work, $O(\log^2 m)$ depth, and $O(m)$ maximum reachable serial space in the PAL model, as previously mentioned. Thus, on the hypercube, this version would take $O((m \log m)/p + \log^2 m \log p)$ time and $O(m + \log^2 m \cdot p \cdot \log p)$ space, with high probability. Implementing algorithms, such as quicksort, directly onto the hypercube results in the same bounds, but is more complicated. Furthermore, we can easily plug the language cost bounds into the cost mappings for other machine models, rather than performing a completely separate analysis.

The central concept of these implementations is executing parallel traversals of the profiling semantics' computation graphs. Previous work showed how to schedule some parallel computation graphs efficiently with respect to time and space [18, 8]. However, this work did not show how these graphs were obtained from or related to more concrete representations of computation, such as a programming language. Our implementations are built directly on this work, but also provide the missing link of showing how our language models relate to their graphs.

[2] This means that these asymptotic bounds hold with arbitrarily high probability—increasing the probability that the bounds hold requires increasing the constant factors of the costs of algorithms.

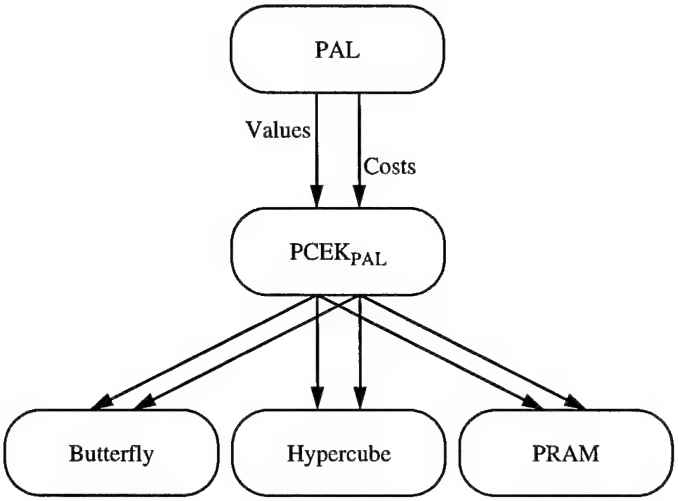


Figure 1.7: Each implementation is staged using an abstract machine of the P-CEK family.

For convenience, we stage each of these implementations by introducing a family of intermediate machine models, one for each of the PAL, PSL, and NESL, as shown in Figure 1.7. The intermediate machine is more abstract than the hypercube, butterfly, and PRAM, as it does not describe the communication network, and it introduces extra control structure. A stack stores states that may each be evaluated in parallel and initially contains a single state representing the entire program before execution. The machine executes a series of steps, where each step

- evaluates states each for unit work and depth;
- creates new states, placing them on the top of the stack; and
- performs any necessary synchronization,

as Figure 1.8 illustrates, and completes when all states have been evaluated. We limit the number of states evaluated per step, so that we can bound the number of states left to evaluate, and thus the space needed for the stack of states. At most $q$ states are evaluated per step, where this number is related to the number of processors, but is sufficiently large to hide communication latency on each of the machine models.

For the PAL and NESL models, we prove that the implementations execute the parallel generalization of depth-first traversals, where each state corresponds to a graph node. Previous results then provide time and space bounds. For the PSL model, the implementations do not execute depth-first $q$-traversals, but only *greedy* $q$-traversals. We can still use previous results to provide bounds for time, but not space.

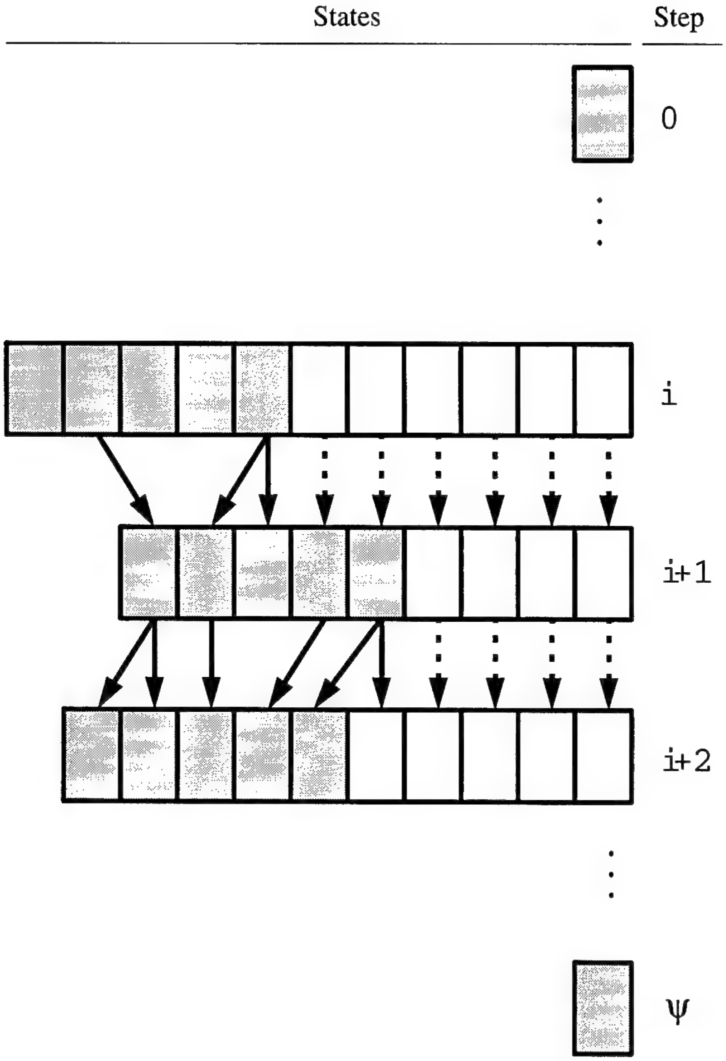


Figure 1.8: Illustration of intermediate machine P-CEK step. It starts with one active state representing the entire program and ends with one active state representing the result value. The states are kept in a stack. At most $q$ states are selected each step. Here, $q = 5$, and these selected states are shaded. These can create zero or more new states (solid arrows). Unselected states are still active in the next step (dashed arrows).

| Relation | Notation |
|---|---|
| NESL is strictly more time-expressive than PAL | NESL $>_{ce}^{PRAM,t}$ PAL |
| PSLp is strictly more time-expressive than PAL | PSLp $>_{ce}^{PRAM,t}$ PSLf |
| PSLf is at least as time-expressive as PAL | PSLf $\geq_{ce}^{PRAM,t}$ PAL |

Figure 1.9: Summary of time-expressiveness of models on a CRCW PRAM. Additional relations are shown in Chapter 10.

The PSL and NESL implementations are asymptotic improvements over their respective existing counterparts:

- Existing implementations of speculative languages all serialize both the suspension and reawakening of sets of threads. Individual sets of suspended threads tend to be small, so that serialization not be a significant problem for many programs. However, it is easy to give examples where this can unnecessarily serialize the bulk of the program computation. So we show how to parallelize these operations, making extensive use of the fetch-and-add operation.

- Our NESL implementation is also an improvement over the existing one in that it is space efficient. The existing implementation executes a level-order traversal of the computation graph, rather than a depth-first $q$-traversal. That may lead to "too much" parallelism, in that it allows the multi-stack of delayed computations to grow asymptotically larger than for the implementation given here.

### 1.2.7 Relating cost models of languages

Once we obtain cost models for languages, this gives a tool for comparing languages on the basis of costs. We first compare some individual algorithms—mergesort, quicksort, and Fast Fourier Transform—in the three models of parallelism. Next, we prove some simulation results between the specific models of interest. Then we generalize these results and define a general notion of *cost-expressiveness* for language models, that relates when one language allows more efficient programs than another. Since language costs are only meaningful in conjunction with its cost mapping to a machine model, cost-expressiveness is also relative to some common machine model. As a simple example, it should not be surprising that the data-parallel model allows asymptotically more efficient programs than the fork-and-join model for most machine models, since the former allows forking of an unbounded number of threads per step. Figure 1.9 summarizes the relative time-expressiveness of the language models used here, assuming the CRCW PRAM as the underlying machine model.

## 1.3 Outline

The remainder of Part I describes the primary areas of related work (Chapter 2) and gives an overview of the notation used (Chapter 3).

Part II describes the methodology of this research using parallel applicative language models (PAL and PAL'). These are the simplest of the models we consider and are appropriate for introducing the framework. Chapters 4 and 5 define the language and its profiling semantics, respectively. Syntactically, the language is based on the $\lambda$-calculus and thus most resembles languages such as Scheme, ML, Haskell, and Id [27, 81, 56, 87]. Chapters 6 and 7 relate the language model to traditional machine models of computation (hypercube, butterfly, and PRAM), staging this via an intermediate model for convenience (P-CEK).

Part III uses this methodology for some other parallel language models. Chapter 8 gives fully and partially speculative models (PSLf and PSLp) for the language. The speculative implementation eliminates a communication bottleneck of existing implementations which can serialize the computation. Chapter 9 extends and modifies the applicative model with sequences and related constructs, using them as the only source of parallelism (NESL). Chapter 10 compares programming in the various models and the asymptotic bounds obtainable in them and introduces the idea of cost-expressiveness.

Finally, Part IV concludes with a summary of the contributions provided.

# Chapter 2

# Related Work

This work lies in the sparsely populated intersection of programming language and complexity theory. Overall, there has been little communication of ideas between these communities, and a meta-goal of this work is to try to build a bridge between these groups. This section discusses not only the work directly related to this research, but outlines some of the other work in this intersection between groups.

## 2.1 Cost models

There has been some work in developing cost models related to those of interest here. However, none of this previous work has been targeted to or fully addresses our goals. Here we present a general overview of the related work—further details are included as relevant in the remainder of the dissertation.

Hudak and Anderson [53] suggested modeling parallelism in functional languages using an extended operational semantics based on partially ordered multi-sets (pomsets). The semantics can be thought of as keeping a trace of the computation as a partial order specifying what had to be computed before what else. Thus, these pomsets correspond closely with computation graphs. Although significantly more complicated, they present semantics (or parts thereof) corresponding to the PAL and PSLf models. However, they did not provide implementations or otherwise relate their model to other models of parallelism or describe how it would effect algorithms.

Roe [105, 106], Flanagan and Felleisen [37], and Moreau [83, 84, 85] provided cost models of speculative evaluation. Roe tracks only the depth of the computation, whereas Flanagan and Felleisen and Moreau track only the work. Roe used his model to analyze algorithms, but did not relate his model to more concrete models. On the other hand, Flanagan and Felleisen and Moreau related their semantics to very abstract machines, but provided no algorithmic analysis.

Blelloch [13, 14] presented NESL with an informal cost model of work and depth, but not space, that is used for algorithmic analysis. Also he did not give a formal cost mapping for NESL's implementation, although he did outline the costs of its mapping to the VRAM model.

Zimmerman [128, 130] introduced a profiling semantics for a data-parallel language for the purpose of automatically analyzing PRAM algorithms. The language therefore almost directly modeled the PRAM by adding a set of PRAM-like primitive operations. Complexity was measured in terms of time and number of processors, as measured for the PRAM. It was not shown, however, whether the model exactly modeled the PRAM. In particular since it is not known until execution how many processors are needed, it is not clear whether the scheduling could be done on the fly.

Goodrich and Kosaraju [44] introduced a parallel pointer machine (PPM), but this is quite different from our models since it assumes a fixed number of processors and allows side effecting of pointers. Abramsky and Sykes [1] introduced the Secd-m machine, which shares a similar basis as our intermediate machines, but is non-deterministic and uses fair merge.

## 2.2 Relating cost models

Previous work on formally relating language-based models (languages with cost-augmented semantics) to machine models is sparse. Jones [60] related the time-augmented semantics of simple while-loop language to that of an equivalent machine language in order to study the effect of constant factors in time complexity. Seidl and Wilhelm [114] provide complexity bounds for an implementation of graph reduction on the PRAM. However, their implementation only considers a single step and requires that you know which graph nodes to execute in parallel in that step and that the graph has constant in-degree. Under these conditions they show how to process $n$ nodes in $O(n/p + p\log p)$ time (which is a factor of $p$ worse than our bounds in the second term).

Riely, Prins, and Iyer [104] defined a data-parallel language model based on Proteus [80] and related it to the VRAM model. The structure of their work is very similar since it is based on earlier versions of this work. Also, their Proteus-based model is similar to the NESL-based model shown here since these two languages are fundamentally similar. However, there is a fundamental difference in implementations and machine models. Here we introduce a machine model with separate domains from that of the language model, whereas they used the same domains for the models. Rather than a cost mapping to relate models, they used a preorder on programs (both uncompiled and compiled programs, since they are in the same domain) based on how efficiently they compute the same function.

Relating cost or complexity models is common in traditional algorithmic and complexity theory. The most widely known examples are probably the comparisons of the many variants of the Turing Machine. One central purpose of such comparisons is to understand what computational constructs add computational power or efficiency to a model. Or viewed from a language perspective, what language features add to a model. Two subclasses of these comparisons are most closely related to this work: those using models of functional language constructs and those using models of parallelism as outlined below. Unlike all of this work, which is driven simply to compare a few models, we also provide a general framework for comparisons of language-based models.

Ben-Amram and Galil [7] described a serial computation model based on pointer-based access to memory (indirect addressing) rather than the usual representation of memory as a giant array (direct addressing). It was to model the core of functional languages such as Lisp, as it included operations such as **car**, **cdr**, **set−car!**, and **set−cdr!** to access and modify the memory. They showed that such models suffered a logarithmic slowdown as compared to traditional direct addressing models in the worst case, as logarithmic time is needed to simulate direct addressing. The models we use follow in this tradition, although we also use arrays in the NESL model. However, our model is based on a high-level language and also incorporate parallelism. But we find the PAL model suffers a corresponding slowdown from the NESL model. Paige [91] also compares models similar to those used by Ben-Amram and Gali, although using the set-based language SETL.

Pippenger [97] also worked with serial pointer-based models, but compared a call-by-value model without side-effects (*i.e.*, without **set−car!** and **set−cdr!**) to a model with these. He found that in general the purely functional model suffers a polylogarithmic slowdown relative to the imperative model. Bird, Jones, and de Moor [103] showed that Pippenger's results extended to also show that the same call-by-value model suffers the same slowdown relative to a purely functional call-by-need model. The implicit side-effect in implementing call-by-need substitutes for the explicit side-effects used by Pippenger.

Other parallel work used the PRAM [38]. While the PRAM is often considered a general model of parallelism useful for designing algorithms, it is also acknowledged as an abstract model which doesn't correspond to an actual machine. It abstractness stems from the unrealistic assumption of constant time communication between arbitrary processors. But the PRAM has been related to other more realistic parallel models, such as those for the butterfly and hypercube [62, 101]. These relations depend entirely on simulating the more realistic communication networks, and for the butterfly and hypercube, these work-efficient simulations entail a slowdown logarithmic in the number of processors. Our cost mappings show a similar difference in bounds between these models, although our bounds on the butterfly and hypercube are not quite the logarithmic factor more needed for the general solution. Other such comparisons of machine models are common, *e.g.*, the implementation of a CRCW or CREW PRAM on an EREW PRAM. This work provides a general framework for such comparisons, although mainly targeted towards the use of more abstract language models of computation.

## 2.3 Implementations of dynamically parallel languages

This section briefly overviews some related work in implementing languages with dynamic parallelism.

Parallel implementations of Id and pH, *e.g.* [4, 86, 94, 93], are generally based on assigning tasks to processors and minimizing the movement of tasks between processors. Each processor has a queue of tasks waiting for a processor. When a processor is not busy, it tries to get a task—it first looks in its own queue, and if its busy, it *steals* a task from another processor's queue. The processor then runs this task until it finishes or it blocks. If it finishes, it reactivates tasks blocked on this one, adding them to appropriate queues of waiting tasks. If it blocks, it adds itself to the appropriate queue of suspended tasks. These implementations attempt to minimize communication, so queues are not implemented in a parallel fashion. This can serialize the entire computation.

The current implementation of NESL is based on flattening nested parallelism to increase granularity. Code using nested sequences compiles to code which only uses sequences of basic datatypes. This creates larger sequences, thus increasing granularity, but at the cost of increasing the cost of some operations. The language is primarily intended for scientific computing where sequences are large. Thus the limitation that only sequence-based operations are parallelized is sufficient. We examine only models of NESL which do not flatten nested parallelism. A flattening model would be somewhat more complicated than the model of Chapter 9 because of the extra compilation step for flattening.

Sisal [34] is an applicative language designed for use on serial and parallel computers. It includes data-parallelism based on a flexible `for` construct combining looping and data reduction on streams of data. It also includes task-parallelism. Streams are non-strict and single-assignment, similar to the I-structures of Id and pH[1]. However, everything else is strict, making synchronization less data-dependent than Id/pH and avoiding overhead such as queues of suspended tasks. Its implementation is based on compiling into a dataflow model and discovering parallelism—what is evaluated in data- or task-parallel depends on a data dependency analysis which may vary between compilers.

Theoretical work by Blumofe and Leiserson [18] and Blelloch, Gibbons, and Matias [8] shows how to efficiently schedule some parallel computations as described by computation graphs. Our implementations and efficiency proofs are built directly on this work, but also provide the missing link of showing how our language models relate to their graphs. Work by Burton [23] and Burton and Simpson [22] also described the space of deterministic parallel models. In particular, for series-parallel computation graphs with constant fan-in and fan-out, they presented a scheduling algorithm using $O(s \cdot p)$ space and within a constant factor of optimal in time for programs with sufficient parallelism. For their work, $s$ is maximum space required by *any* serial DFT of the graph, rather than the space required by the standard DFT that traverses the ready nodes in left-to-right order.

## 2.4 Language models and their uses

Formally or informally, models of language run-time costs have been used in a number of areas. This section describes some of these areas where a language model is particularly important.

[1] An I-structure is an array of single-assignment locations.

### 2.4.1 Automatic complexity analysis

There have been several approaches to automatically deriving complexity bounds for programs. The basic idea of these is to convert a program into a set of recurrence equations defining the costs, and then solve the recurrence equations. Both steps depend on identifying some *ad hoc* collection of general recurrence patterns. To create these equations involves identifying appropriate *size abstractions* of the data, such as the length of a list or the depth of a tree. Most of this work has been restricted to worst-case analysis of serial functional languages [124, 76, 77, 78, 109, 110, 111, 115, 108, 35, 126, 128, 127, 129] to simplify the analysis of recurrences. However, there has also been some work on worst-case analysis of for a PRAM-like parallel language [130], defining the depth of the computation and the maximum number of processes that the computation can employ. Also there has been some work on worst-case analysis of imperative languages [28] and average-case analysis of serial functional languages [36].

Converting the program into cost recurrence equations requires at least an informal definition of a language's costs, although some used formal definitions. The framework presented here provides formal definitions of costs which could be used for automatic complexity analysis, plus a cost mapping to relate the analyzed bounds to more realistic machine models or to traditional complexity theory.

### 2.4.2 Compiler analysis

Any compiler optimization technique which analyzes the cost of code uses at least an informal cost model. Any such technique should be based on a formal model, such as provided by this work, so that the resulting optimization can be verified and quantified. For example, Knopp [67, 68] and Flanagan and Felleisen [37] both used language models somewhat similar to the PAL model in analyses to avoid runtime checks.

### 2.4.3 Profiling tools

A profiling tool (or profiler) instruments source or object code to keep track of run-time costs. It is used for run-time debugging and performance analysis and for guiding optimization. Any profiler requires at least an informal definition of a language's costs, but frequently these definitions are *ad hoc* or special purpose. Some recent profiling tools have been based on the language semantics and a more formal notion of the costs [66, 65, 112, 113]. Since profilers generally need to produce highly accurate resource profiles, they require more detailed semantics than the abstract semantics provided here. But our framework allows detailed semantics and provides a formalism which could be used as the basis for these semantics-based profilers.

## 2.5 Provably correct implementations

The general goal in provably correct implementation is obtaining compilers which produce efficient and provably correct code, *e.g.* [95, 71, 98, 24, 19]. At the core, this work provides a relatively abstract source code semantics, a very detailed object code semantics, and a provably correct compiler mapping between the two. These correspond to our language model, machine model, and mapping. Since most are interested in obtaining realistically efficient compilers, they use more complicated machine models and mappings. But they do not give any bounds on compiler efficiency.

## 2.6 Other mixes of language theory and algorithmic theory

The following work is also in a broadly defined intersection between programming language theory and algorithmic theory, although not directly related to this work.

This dissertation is an example of intensional semantics, as it formally describes not only in *what* a computation results, but also *how* the computation proceeds. But most of the work in intensional semantics concentrated on areas such as full abstraction (proving denotational and operational semantics equivalent) or traces of concurrent processes [21]. However, Gurr gave a categorical framework for defining language cost models and generalizing these to asymptotic complexity models [49]. While powerful, his framework does not correspond to typical programmers' intuition because of its use of category theory. And while very intriguing, the generalization to complexities is only partially successful. Also, Talcott provided an intensional theory similar to those used for automatic cost analysis [119].

Jones, *e.g.*, [60] has been exploring traditional complexity theory from a programming language perspective. This includes re-examining how certain complexity classes arise from different language idioms. This perspective results in some novel results, including a theoretical equivalent of the intuition that constant factors in performance really do matter in what can be computed.

*Skeletons* are parameterized complexity functions obtained using traditional algorithm analysis techniques, *e.g.*, for a general-purpose divide-and-conquer algorithm [118]. Work in this area also uses functional languages for simplicity. We use the same basic idea to parameterize our bounds with respect to the load-balancing and latency costs in various machine models. Note that one general technique used with skeletons is *shape analysis* which is a generalization of the size abstraction in automatic complexity analysis. Also, the work of Skillicorn, *et al.* overlaps with that of automatic complexity analysis [25, 116, 117]. In addition, their work also overlaps greatly with that in algorithm analysis using high-level functional data parallel languages, which also includes work using NESL, *e.g.*, [11, 10, 46], and other work in the Bird-Meertens formalism.

There has been some work on obtaining asymptotically efficient data structures using functional languages, *e.g.* [52, 30, 89]. These each approached traditional algorithmic analysis problems, but from the perspective of modern programming languages, using language features such as higher-order functions and laziness. Many problems of interest here involve persistent data structures, where updates do not destroy the original data structure. We use some simple persistent data structures, *e.g.*, for environments. So far, all of this work has been for serial languages, but such work could serve as prime examples for analysis in our framework, given the appropriate language models.

## 2.7 Expressiveness

Discussions of programming languages often include informal discussions of how they are more *expressive* than previous languages. However, there is no single formal notion of what this means. For example, Felleisen [32] and Mitchell [82] compared languages based on different criteria—Felleisen observed when language features could equivalently be defined as macros, whereas Mitchell observed when features can be used as abstraction contexts. Both also discussed additional previous work on comparing languages.

Much of this work starts with the basic realization that the most fundamental comparison, that of what functions the languages can compute, is not a fine enough distinction. Most useful languages are Turing-equivalent, and thus equivalent under that comparison. The notion of cost-expressiveness defined here is another way to compare languages. Since it is based on intensional aspects of the language, whereas previous comparisons are based on extensional aspects, it is an orthogonal comparison that can be combined with any of the previous ones.

# Chapter 3

# Notation

This chapter serves as a reference chart of the notation used here. The notation will be explained further as introduced in the following chapters.

Note that some notation, especially the semantic domain meta-variables, is overloaded between the various language models. The general purpose meta-variable $X$ represents notation that varies between the language and machine models. Which definition is appropriate should always be clear from context. Also, notationally we do not distinguish arrays, ordered sets, stacks, and queues. Some additional symbols are used for purely local definitions.

This notation is not identical to that used in earlier presentations of this work, as many details have been reworked for consistency between models and for overall clarity.

| Meta-variables: Primes and subscripts are used to obtain additional meta-variables in a given domain. Numerical subscripts are also used to denote indexing of array components. | |
|---|---|
| $b$ | Boolean constant |
| $c$ | Syntactic constant |
| $C$ | Control information of a state |
| $D$ | Semantics derivation |
| $d$ | Depth cost |
| $e$ | Syntactic expression |
| $E$ | Set of computation graph edges |
| $g$ | Computation graph |
| $gc$ | Number of garbage collections during evaluation |
| $i, j, k, m$ | Integer |
| $I$ | Intermediate state |
| $l$ | Location |
| $n$ | Computation graph node |
| $ns$ | Computation graph source node |
| $nt$ | Computation graph sink node |
| $N$ | Set of computation graph nodes |
| $NE$ | Mapping from nodes to their children (ordered set of nodes) in a computation graph |
| $p$ | Total number of processors |
| $P$ | Computation graph traversal prefix |
| $q$ | Maximum number of selected states each step |
| $Q$ | Number of states processed |
| $R$ | Set of root values |
| $s$ | Space cost |
| $st$ | State |
| $St$, $StA$ | Array, multi-stack, *etc.* of states or active states |
| $sv$ | Store-value |
| $t$ | Time cost |
| $T$ | Computation graph traversal |
| $v$ | Value |
| $V$ | Set of visited nodes in computation graph traversal |
| $w$ | Work cost |
| $x, y, z, \_$ | Variable |
| $\rho$ | Environment (mapping from variables to values) |
| $\sigma$ | Store (mapping from locations to store-values) |
| $\tau$ | Thread |
| $\psi$ | Number of evaluation steps |
| – | Wildcard in meta-syntactic pattern matching |

| Expressions: Not all of these expressions are used in each model. | |
|---|---|
| $\lambda x.e$ | Abstraction (user-defined function) with bound variable $x$ and body $e$ |
| $e_1\ e_2$ | Application of function $e_1$ and argument $e_2$ |
| $(e_1,e_2)$ | Pair |
| $\textbf{let}\ x = e_1\ \textbf{in}\ e_2$ | Non-recursive local binding of the value of $e_1$ to variable $x$ |
| $\textbf{letrec}\ x\ y = e_1\ \textbf{in}\ e_2$ | Recursive local binding of a function named $x$ with argument $y$ and body $e_1$ |
| $\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3$ | Conditional |
| $\{e' : x\ \textbf{in}\ e\}$ | "For-each" expression evaluating $e'$ for each binding of $x$ to an element of $e$ |
| $@\ v_1\ v_2$ | Application of function $v_1$ and argument $v_2$ |
| $\textbf{done}\ v$ | End of computation with result $v$ |
| $\Sigma\ l_1\ l_2\ l'\ i$ | Add $i$th element of sequence at $l_1$ into result sequence at $l_2$ with current running total at $l'$ |
| $FV(e)$ | Set of free variables of expression $e$ |

| Values and Store-values: Not all of these are used in each model. | |
|---|---|
| $\textbf{cl}(\rho,x,e)$ | Closure (function) with argument $x$ |
| $\textbf{cl}(\rho,x,y,e)$ | Recursive closure named $x$ with argument $y$ |
| $\langle v_1,v_2\rangle$ | Pair |

| Continuations: Not all of these are used in each abstract machine. | |
|---|---|
| $\bullet$ | Empty or initial continuation |
| $\mathsf{fun}\langle X\ \kappa\rangle$ | Continuation marking function branch of application |
| $\mathsf{arg}\langle X\ \kappa\rangle$ | Continuation marking argument branch of application |
| $\mathsf{end}\langle l\ \kappa\rangle$ | Continuation marking each branch of for-each |
| $\mathit{throw}(v,\kappa)$ | Throw value $v$ to continuation $\kappa$ |

| Mappings: | |
|---|---|
| $\cdot$ | Empty mapping |
| $X[X_d \mapsto X_r]$ | Mapping $X$ extended with the binding of $X_d$ to $X_r$. $X_d$ may occur in $X$ |
| $X(X_d)$ | Item bound to domain element $X_d$ in the mapping $X$, or set of items bound to set of domain elements in the mapping |
| $X \cup X'$, $\bigcup_{i=0}^{m-1} X_i$ | Union of mappings (disjoint domains) |
| $X \sqcup X'$ | Union and update of mappings ($X'$ may replace bindings in $X$) |
| $X \uplus X'$ | Union of computation graph adjacency lists |
| $dom(X)$ | Domain of mapping $X$ |
| $rng(X)$ | Range of mapping $X$ |

| Arrays and other data structures: We use the same notation for arrays, ordered sets, queues, and multi-stacks. | |
|---|---|
| $[X_0, \ldots, X_{n-1}]$ | A homogeneous data structure |
| $X[X'/i]$ | New data structure replacing $i$th element of $X$ with $X'$ |
| $\vec{X}$ | A homogeneous data structure $[X_0, \ldots, X_{\lvert\vec{X}\rvert}]$ |
| $\lvert X \rvert$ | Number of elements in data structure $X$ |
| $\vec{X} \mathbin{+\!\!+} \vec{X}'$, $\mathop{+\!\!+}_{i=0}^{m-1} X_i$ | Combining (*e.g.*, appending, pushing, enqueuing) of data structures |

| Evaluation: | |
|---|---|
| $X_c \vdash e \xrightarrow{X} v; X_o$ | Evaluation in semantics $X$ starting in context $X_c$ with $e$ results in $v$ and cost information $X_o$. The forms of $X_c$ and $X_o$ depend on the semantics. |
| $\overset{X}{\hookrightarrow}_{X'}$ | Transition $X'$ used on each applicable state in a substep of intermediate machine $X$. |
| $StA, \sigma \overset{X,q}{\hookrightarrow}_k StA', \sigma'; X_o$ | $k$ steps of the intermediate machine $X$, selecting at most $q$ states per step. The machine starts with active states $StA$ and store $\sigma$ and ends with active states $StA'$, store $\sigma'$, and cost information $X_o$. |
| $e \vdash e \overset{X,q}{\Longrightarrow} v; X_o$ | Evaluation in intermediate machine $X$, selecting at most $q$ states per step, of expression $e$ results in $v$ and cost information $X_o$. |
| $\delta(c, v)$, $\delta(X_c, c, v)$ | Application of $c$ to $v$, possibly requiring a context $X_c$. |

| Costs of evaluation: | |
|---|---|
| $\mathbf{1}$ | Singleton node computation graph |
| $\mathbf{1}\leftarrow g$ | Singleton node and edge from the minimum sink of $g$ |
| $g_1 \oplus g_2$ | Serial combination of computation graphs |
| $g_1 \otimes g_2$ | Parallel fork-and-join combination of computation graphs |
| $\bigotimes_{i=0}^{m-1} \vec{g}$ | Parallel fork-and-join combination of computation graphs |
| $g_1 \wedge g_2$ | Forking of computation graphs |
| $S_n(n)$ | Net space allocation of node $n$ that is independent of traversal |
| $D(g)$ | (Minimum) Depth cost of a computation graph $g$ |
| $D'(g)$ | Maximum depth cost of a PSL computation graph $g$ |
| $S(R, \sigma)$ | Space reachable in store $\sigma$ from roots $R$ |
| $S_n(n)$ | Space of computation graph node $n$ |
| $S_P(P)$ | Space of traversal prefix $P$ |
| $S_T(T)$ | Space complexity of traversal $T$ |
| $TF(p)$ | Time cost of fetch-and-add operation on $p$ processors |
| $TS(p)$ | Time cost of scan and reduce operations on $p$ processors |
| $W(g)$ | Work cost of a computation graph $g$ |

| States: Not all of these are used in each abstract machine. | |
|---|---|
| $(e, \rho, \kappa)$ | State |
| $\mathsf{@stub}\langle l\ i\ k\ \kappa\rangle$ | Stub state representing the $k-i$ states having expressions @ $l$ $i$ through @ $l$ $(k-1)$, an empty environment, and continuation $\kappa$ |
| $\mathsf{\Sigma stub}\langle l_1\ l_2\ l'\ i\ k\ \kappa\rangle$ | Stub state representing the $k-i$ states having expressions $\Sigma$ $l_1$ $l_2$ $l'$ $i$ through $\Sigma$ $l_1$ $l_2$ $l'$ $(k-1)$, an empty environment, and continuation $\kappa$ |

| Intermediate states: Not all of these are used in each abstract machine. | |
|---|---|
| $\mathsf{Fin}\langle X\ X'\rangle$ | State finishing this step |
| $\Sigma\langle l_1, l_2, l', i, \kappa\rangle$ | Add $i$th element of sequence at $l_1$ into result sequence at $l_2$ with current running total at $l'$, continuing with the continuation $\kappa$ if this is the last element |

| Other functions and relations: | |
|---|---|
| $T_X[\![X']\!]$, $T_X[\![X'']\!]_{\sigma'}[\![X']\!]_{\sigma}$ | Translations to model $X$ |
| $X_1 \geq_{ce}^{X,c} X_2$ | Language model $X_1$ is at least as cost-expressive as $X_2$, when both are implemented on machine model $X$, for implementation cost measure $c$. |

# Part II

# Methodology

# Chapter 4

# Language

In this dissertation we are interested in parallel languages with first-class higher-order functions. As discussed in Section 1.1, these languages are

- abstract, and thus easy to analyze extensionally, but generally not well understood intensionally; and
- general, and thus representative of many characteristics of realistic languages.

This part of the dissertation uses a simple model called the Parallel Applicative $\lambda$-Calculus (PAL) as an introductory example. While this chapter is primarily concerned with the syntax of the language, which uses standard $\lambda$-calculus notation, it also anticipates the formal semantics and implementation with some informal discussion of the parallelism in the model. In particular, its semantics is based on the standard call-by-value (applicative-order) evaluation strategy.

Many features of modern languages (*e.g.*, data structures, conditionals, recursion, and local variables) can be simulated in the standard $\lambda$-calculus with constant overhead, therefore not affecting asymptotic performance. Thus, we use relatively small and simple languages, especially in this overview of the methodology. This eases the description of the languages and the proofs of simulation results (*e.g.*, Chapter 7). In particular, Section 4.1 defines the PAL model with just the basic $\lambda$-calculus with a few constants, a minimal language for the first examples. Since that language is too sparse for reasonable examples, Section 4.2 introduces an extended $\lambda$-calculus that includes a representative sample of features of modern languages: pairs, explicit recursion, conditionals, and a larger selection of constants. Chapter 5 shows that for the purpose of asymptotic performance bounds, these two languages are equivalent, *i.e.*, that simulating the extensions in the smaller $\lambda$-calculus requires only constant overhead.

All of the languages presented in this dissertation are untyped. We define the costs of evaluation only for those programs which correctly execute, independent of the static checking that types would provide. And while types are useful for optimizing the program implementation, this information does not provide the asymptotic benefits we are looking for.

$$\begin{array}{lclcll}
c & \in & \mathit{Constants} & & & \\
x, y, z & \in & \mathit{Variables} & & & \\
e & \in & \mathit{Expressions} & ::= & c \mid x \mid & \\
 & & & & \lambda x.e \mid & \text{abstraction} \\
 & & & & e_1\ e_2 & \text{application}
\end{array}$$

Figure 4.1: Basic λ-calculus expressions.

## 4.1 λ-calculus

The basic λ-calculus consists of the following kinds of expressions:

- constants $c$, such as numbers or primitive functions like addition;
- variables $x$;
- abstractions $\lambda x.e$, which represent user-defined functions with bound variable $x$ and body $e$; and
- applications $e_1\ e_2$, *i.e.*, function calls with function $e_1$ and argument $e_2$.

All functions are *first-class*, which means that they can be passed as arguments or returned from functions like any other value. Furthermore, the function of an application expression can be *any* expression, not just an identifier, with the assumption that it evaluates to something appropriate.

The core syntax of the λ-calculus is given in Figure 4.1, where the set of constants is defined later, and the set of variables is countably infinite. The free variables of an expression, $FV(e)$, are defined as usual, where abstraction is the only variable binding construct.

For an application expression $e_1\ e_2$ in the call-by-value (applicative-order) λ-calculus, the function $e_1$ and argument $e_2$ can always be evaluated in parallel, since the argument is always evaluated anyway and there are no side-effects to make the relative order of evaluation noticeable. There is a choice of how to synchronize these parallel computations. The PAL model of Chapter 5 synchronizes after both the function and argument finish evaluating, whereas the PSL model of Chapter 8 synchronizes only when the argument's value is needed.

### Arithmetic constants

Without arithmetic constants, the λ-calculus is not a practical model of realistic languages. Numerous schemes exist for encoding numerals in the λ-calculus. At best, these allow constant time successor, predecessor, and zero-test operations and polylogarithmic time for other

$$\begin{array}{lllll} i & \in & \mathit{Integers} & & \\ c & \in & \mathit{Constants} & ::= & i \mid \mathbf{add} \mid \mathbf{sub} \mid \mathbf{mul} \mid \mathbf{lt} \mid \qquad \text{numeric constants} \\ & & & & \mathbf{add}_i \mid \mathbf{sub}_i \mid \mathbf{mul}_i \mid \mathbf{div}_i \mid \mathbf{lt}_i \end{array}$$

Figure 4.2: Basic λ-calculus constants.

operations [92]. But since most machines use a fixed-precision arithmetic, they provide constant time operations for multiplication, division, equality, *etc.* To model this, we include such operations as constants.

There exist encoding schemes to encode groups (*e.g.*, arrays) of integers as a single integer and operations on such groups as functions on their encodings. However, if we include division along with a standard set of operations on arbitrary-precision integers, the integers are *compressible* [7], which means that such encodings can be asymptotically faster than the operations on the raw groups of data. To avoid this we restrict the set of integers operations to include, *e.g.*, division by two, rather than full division.[1]

To keep the number of syntactic forms, and thus the number of semantic rules and the number of proof cases, to a minimum, we do not syntactically distinguish unary and binary functions. Instead, a binary function such as addition takes its arguments one at a time, *i.e.*, $\mathbf{add}\ 1\ 2 \equiv (\mathbf{add}\ 1)\ 2$. Thus the core language uses the constants defined in Figure 4.2. For example, **add** represents binary addition, and $\mathbf{add}_i$ represents the unary addition of the number $i$. While the unary functions are redundant with the corresponding binary functions, we include them in the syntax of expressions to simplify the presentation of the semantics (the unary functions are the result of applying the corresponding binary functions to their first argument) and for syntactic consistency with the extended λ-calculus.

## 4.2 Extended λ-calculus

The basic λ-calculus of the previous section is too sparse for even small examples, so we introduce a language with some extensions. We add pairs to have a better way to allow $n$-ary constant functions. We add boolean constants, conditional branching, and explicit recursive function bindings as other typical extensions. Our extended language syntax is then defined in Figure 4.3. The expression $\mathbf{letrec}\ x\ y = e_1\ \mathbf{in}\ e_2$ creates a possibly recursive function $x$ with bound variable $y$ and function body $e_1$. Since this is a recursive binding, this binds both $x$ and $y$ in the function body. The entire function definition is bound in the program body $e_2$. Again, each of the constants has the obvious intended meaning. Also the free variables of an expression are defined as usual, where abstraction and **letrec** are the only binding constructs.

[1] Another option would be to bound the range of the integers, as in most languages. This would be a better solution for a more detailed and accurate semantics.

| | | | | | |
|---|---|---|---|---|---|
| $i$ | $\in$ | *Integers* | | | |
| $b$ | $\in$ | *Booleans* | $::=$ | **true** \| **false** | |
| $c$ | $\in$ | *Constants* | $::=$ | $i$ \| **add** \| **sub** \| **mul** \| **div**$_i$ \| | numeric constants |
| | | | | **lt** \| **eq** \| **gt** \| | |
| | | | | $b$ \| **and** \| **or** \| **not** \| | boolean constants |
| | | | | **fst** \| **snd** | pair constants |
| $x, y, z$ | $\in$ | *Variables* | | | |
| $e$ | $\in$ | *Expressions* | $::=$ | $c$ \| $x$ \| | |
| | | | | $\lambda x.e$ \| | abstraction |
| | | | | $e_1\ e_2$ \| | application |
| | | | | $(e_1,e_2)$ \| | pairing |
| | | | | **letrec** $x\ y = e_1$ **in** $e_2$ \| | recursive function binding |
| | | | | **if** $e_1$ **then** $e_2$ **else** $e_3$ | conditional branching |

Figure 4.3: Extended $\lambda$-calculus.

These extensions are only representative of what a realistic language would include. For example, other local binding expressions, data structures, and pattern matching can be simulated with **letrec**, pairing, and conditionals. This syntax is sufficient for reasonable examples while still simple enough for small semantic definitions of the language models. For readability, we will assume such further extension in Chapter 10.

As we show, the encoding of pairs in the $\lambda$-calculus allows the two expressions to be evaluated in parallel in the PAL model, so the intension is to have applications and pairs be the only sources of parallelism in this model. Furthermore, the encoding of conditionals respects the usual serialization of the test and branch.

Note that we overload notation, *e.g.*, reusing meta-variables between the languages, so as not to clutter the notation. This is common throughout the dissertation. Which definition is intended should always be clear from context.

# Chapter 5

# Profiling semantics

Recall that most semantics define only the *extensional* properties of a language: a program's results and termination properties. A profiling semantics augments such an extensional semantics with definitions of the *intensional* information, the costs of evaluating the expression [108, 110]. In particular we are interested in the time and space costs of evaluation, where parallel time is modeled by *work* and *depth*, or more generally, *computation graphs.*

We add these costs to operational semantics in a natural deduction, "big-step", operational style of semantics. Any style of semantics, including denotational, could be used, but we use this style because we find it more convenient. In particular, an operational style is more suited to

- defining the costs incurred during evaluation, because it describes the process of evaluation;
- defining machine models traditionally defined by a state transition relation; and
- proving equivalences with such machine models.

A "big-step" semantics directly defines the value of an expression, as opposed to a "small-step" semantics which first defines what an expression reduces to in one step, and then defines the value of the expression to be the transitive closure of this reduction. A "big-step" semantics hides evaluation details which are irrelevant for the language models we use here.

Section 5.1 reviews the call-by-value operational semantics of the basic $\lambda$-calculus. Section 5.2 explains computation graphs and defines their form for the PAL model. Section 5.3 then extends this operational semantics to a profiling semantics defining the computation graphs for the PAL model. Section 5.4 further extends the profiling semantics to define the space usage of evaluation. These semantics use the basic $\lambda$-calculus for simplicity. But Section 5.5 shows that basing the PAL model on the extended $\lambda$-calculus instead affects time and space bounds by only a constant factor.

$$
\begin{array}{lllll}
\rho & \in & \mathit{Environments} & = & \mathit{Variables} \overset{\mathit{fin}}{\rightarrow} \mathit{Values} \\
v & \in & \mathit{Values} & ::= & c \mid \\
 & & & & \mathbf{cl}(\rho,x,e) \qquad\qquad \text{closure}
\end{array}
$$

Figure 5.1: The definition of call-by-value λ-calculus run-time domains.

## 5.1 Call-by-value λ-calculus semantics

This section provides a brief review of a standard way to define the call-by-value λ-calculus semantics.

An operational semantics defines the result value of evaluating an expression. In the form of operational semantics used throughout this dissertation, evaluation is performed in the context of an *environment*, a mapping from a finite set of variables to their values. (See Chapter 3 for notation used with mappings.) The values resulting from the basic λ-calculus are the constants we have included and *closures*, the representation of functions. Thus, environments and values are defined mutually recursively as in Figure 5.1.

Semantics are defined recursively in terms of a *judgment*, or relation, describing the evaluation of expression $e$ to value $v$. A standard call-by-value semantics is often defined as in Definition 5.1. In each semantics, we annotate the arrow with the semantics name for clarity. We are primarily interested in the "top level" case where the evaluation starts with an empty environment, *i.e.*, $\cdot \vdash e \xrightarrow{\lambda} v$ represents the evaluation of program $e$ to its result value $v$.

**Definition 5.1 (Call-by-value λ-calculus semantics)** *In the call-by-value λ-calculus,* in the environment $\rho$, the expression $e$ evaluates to value $v$, *or*

$$\rho \vdash e \xrightarrow{\lambda} v,$$

*if it is derivable from the rules*[1] *of Figure 5.2. Figure 5.3 defines the $\delta$ function for the application of constants.*

As usual, a constant evaluates to itself, an abstraction evaluates to a *closure* containing the current environment, and a variable evaluates to the value found for that variable in the current environment. An application evaluates each of the function and argument and

- if the function value is a closure, it evaluates the closure body using the closure's defining environment; or
- if the function value is a constant, it evaluates the constant application as defined by $\delta$.

[1] Items above each horizontal line are assumptions needed for the judgment below the line to hold. Rules without a horizontal line are axioms.

$$\rho \vdash c \xrightarrow{\lambda} c \qquad \text{(CONST)}$$

$$\rho \vdash \lambda x.e \xrightarrow{\lambda} \mathbf{cl}(\rho,x,e) \qquad \text{(LAM)}$$

$$\frac{\rho(x) = v}{\rho \vdash v \xrightarrow{\lambda} v} \qquad \text{(VAR)}$$

$$\frac{\rho \vdash e_1 \xrightarrow{\lambda} \mathbf{cl}(\rho',x,e_3) \qquad \rho \vdash e_2 \xrightarrow{\lambda} v_2 \qquad \rho'[x \mapsto v_2] \vdash e_3 \xrightarrow{\lambda} v}{\rho \vdash e_1\ e_2 \xrightarrow{\lambda} v} \qquad \text{(APP)}$$

$$\frac{\rho \vdash e_1 \xrightarrow{\lambda} c \qquad \rho \vdash e_2 \xrightarrow{\lambda} v_2 \qquad \delta(c, v_2) = v; g_3}{\rho \vdash e_1\ e_2 \xrightarrow{\lambda} v} \qquad \text{(APPC)}$$

Figure 5.2: Call-by-value operational semantics using the basic λ-calculus and the definition of $\delta$ in Figure 5.3.

| $c$ | $v$ | $\delta(c, v)$ | if/where |
|---|---|---|---|
| $\mathbf{add}$ | $i_1$ | $\mathbf{add}_{i_1}$ | |
| $\mathbf{add}_{i_1}$ | $i_2$ | $i_1 + i_2$ | |
| $\mathbf{sub}$ | $i_1$ | $\mathbf{sub}_{i_1}$ | |
| $\mathbf{sub}_{i_1}$ | $i_2$ | $i_1 - i_2$ | |
| $\mathbf{mul}$ | $i_1$ | $\mathbf{mul}_{i_1}$ | |
| $\mathbf{mul}_{i_1}$ | $i_2$ | $i_1 * i_2$ | |
| $\mathbf{div}_{i_1}$ | $i_2$ | $\lfloor i_1 / i_2 \rfloor$ | |
| $\mathbf{lt}$ | $i_1$ | $\mathbf{lt}_{i_1}$ | |
| $\mathbf{lt}_{i_1}$ | $i_2$ | $\mathbf{cl}(\cdot,x,\lambda y.x)$ | $i_1 < i_2$ |
| $\mathbf{lt}_{i_1}$ | $i_2$ | $\mathbf{cl}(\cdot,x,\lambda y.y)$ | $i_1 \geq i_2$ |

Figure 5.3: The $\delta$ function defining constant application for the call-by-value λ-calculus.

$$\dfrac{[x \mapsto \mathbf{cl}(\cdot,y,1)](x) = \mathbf{cl}(\cdot,y,1)}{[x \mapsto \mathbf{cl}(\cdot,y,1)] \vdash x \xrightarrow{\lambda} \mathbf{cl}(\cdot,y,1)}\text{(VAR)} = D_2$$

$$\dfrac{\dfrac{}{\cdot \vdash \lambda x.x \xrightarrow{\lambda} \mathbf{cl}(\cdot,x,x)}\text{(LAM)} \qquad \dfrac{}{\cdot \vdash \lambda y.1 \xrightarrow{\lambda} \mathbf{cl}(\cdot,y,1)}\text{(LAM)} \qquad D_2}{\cdot \vdash (\lambda x.x)\ (\lambda y.1) \xrightarrow{\lambda} \mathbf{cl}(\cdot,y,1)}\text{(APP)} = D_1$$

$$\dfrac{D_1 \qquad \dfrac{}{\cdot \vdash 2 \xrightarrow{\lambda} 2}\text{(CONST)} \qquad \dfrac{}{[y \mapsto 2] \vdash 1 \xrightarrow{\lambda} 1}\text{(CONST)}}{\cdot \vdash (\lambda x.x)\ (\lambda y.1)\ 2 \xrightarrow{\lambda} 1}\text{(APP)}$$

Figure 5.4: Call-by-value $\lambda$-calculus operational semantics derivation for Example 5.1. For readability, the derivation tree is broken into three subtrees.

Using the APPC rule with the $\delta$ function is a convenient way to define the application of most constants—in particular, those that do not depend on the general evaluation relation. Alternatively, we could define a separate semantic application rule for each constant.

Since this simple language does not include explicit booleans for the result of the less-than test, a programmer would use standard encodings of booleans instead. Section 5.5 uses a $\lambda$-calculus with additional constructs, including explicit booleans and also reviews how the encodings used here represent booleans.

Division by zero does not cause an error here since, for simplicity, we do not include errors in the semantics. Instead, we have two choices:

- to have a "not a number" value as in the IEEE standard; or
- to leave division by zero, and any program evaluation causing it, undefined.

**Example 5.1** *As a small example of a operational semantics derivation, observe the evaluation of* $(\lambda x.x)$ $(\lambda y.1)$ 2. *The derivation forms a tree, as Figure 5.4 shows.*

The semantics can be read as a simple interpreter for the language. For better efficiency, many compilation techniques have been developed over the years for the $\lambda$-calculus and related languages. The implementations used here are in the tradition of the SECD state machine of Landin [70] and its descendants, as further described in Chapter 6.

## 5.2 Computation graphs

We use directed acyclic graphs (DAGs) to represent the process of computation, generalizing the computation's work and depth. As usual, a directed graph consists of a set of nodes and a set of directed edges. Each node of a computation graph represents a single unit of computation: unit work and unit depth. An edge represents a control or data dependency, where the *child* depends on the *parent*. Additionally, we place a total order on the children of each node to distinguish the scheduling order on the nodes. Thus, more accurately, these are *ordered DAGs*.

We use computation graphs for several related purposes:

- They provide a way to visualize computation and to gain an intuitive understanding about these models.

- They formally generalize the work and depth costs of a computation in that the work and depth can be easily defined in terms of the computation graph $g$. The work $W(g)$, is the number of nodes in graph $g$, and the depth $D(g)$ is the number of levels, or equivalently, one more than the length of the longest path, in graph $g$.

- They are useful to describe the scheduling of computation on machine models (*cf.* Chapters 6 and 7). For example, we describe how to schedule nodes with the constraint that all parents must be scheduled prior to their children.

We will use computation graphs for all models in this dissertation, although the more detailed structure of the graphs varies among models.

Figure 5.5 illustrates the computation graphs for the PAL model. We can distinguish two nodes (potentially the same node) in the subgraph for any subexpression's evaluation: its source *ns* and sink *nt*. The source represents the start of a computation, and the sink represents its end. We draw graphs as a diamond with its source at the top and its sink at the bottom. Evaluating a constant, variable, abstraction, *etc.*, requires only constant work and depth, and the computation graph is a single node. Evaluating an application $e_1\ e_2$ introduces parallelism—nodes before and after the parallel branches represent the initiation and synchronization associated with this parallelism.

We choose a consistent ordering on the evaluation of subexpressions, arbitrarily placing the function's subgraph before that of the argument. This choice makes the branching in computation graphs resemble that of the corresponding syntax tree. As explained later, this ordering will reflect the execution ordering when there are not enough processors to parallelize all computation. Within the context of the example languages presented here, the choice of ordering is irrelevant.

The reader may be concerned over the form of these graphs. First, the nodes may represent significantly different amounts of real work, as for example, synchronization would take significantly more time than evaluating a constant. Second, either or both of variable lookup or closure creation (*i.e.*, evaluating a variable or a function) could take non-constant time, but they are both represented by single nodes. But the computation graphs represent the time costs as defined abstractly in the language model, and these costs must still be mapped to the machine model. For example, later chapters show that in the given implementations, the latency of communication for synchronization and variable lookup requires up to logarithmic

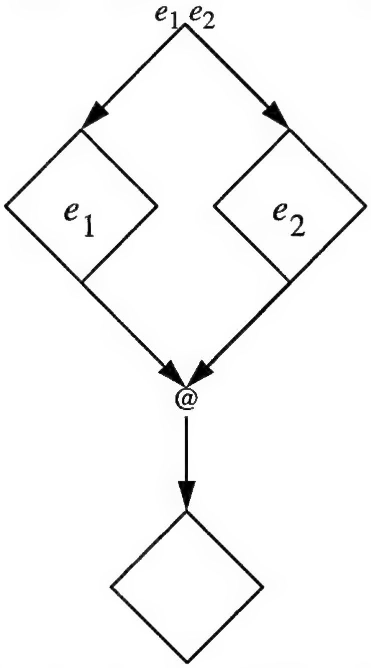
| Expression $e$: | $c$, $x$, or $\lambda x.e$ | $e_1\ e_2$ |
|---|---|---|
| Graph $g$: | ● |  <br>where the last subgraph is for either the body of the user-defined function (closure) or the application of the constant to which $e_1$ evaluates |

Figure 5.5: Illustration of computation graphs for the PAL model.

(in number of processors) time, in the machines of interest. The model here represents the following ideas:

- the programmer does not need to know why this logarithmic factor is necessary, but that it is part of the overhead of the implementation; and
- the implementor is not constrained to where to introduce this overhead.

### PAL **computation graphs formalized**

The computation graphs of the PAL model and its variants are all *single-source, single-sink computation graphs*, as defined in Definition 5.2.

**Definition 5.2** *A* single-source, single-sink computation graph *is a triple $(ns, nt, NE)$ of the source, the sink, and the mapping of nodes to their children, such that the mapping induces an acyclic partial ordering. This mapping represents the edges of the graph in the form of adjacency lists, and the nodes of the graph are implicitly represented in these edges.*

*Mappings representing sets of adjacency lists are combined with $\uplus$:*

$$NE_1 \uplus NE_2 = \bigcup_{n \in N} \begin{cases} [n \mapsto NE_1(n)] & n \in dom(NE_1), n \notin dom(NE_2) \\ [n \mapsto NE_2(n)] & n \notin dom(NE_1), n \in dom(NE_2) \\ [n \mapsto NE_1(n) \mathbin{+\!\!+} NE_2(n)] & n \in dom(NE_1) \cap dom(NE_2) \end{cases}$$

*where $N = dom(NE_1) \cup dom(NE_2)$. When $NE_1$ and $NE_2$ are guaranteed not to contain any of the same sources, i.e., they have disjoint domains, $NE_1 \uplus NE_2 = NE_1 \cup NE_2$.*

In the PAL model, all computation graphs are of *serial-parallel* form, as defined in Definition 5.3.

**Definition 5.3** *A* series-parallel graph $g$ *is a single-source, single-sink graph formed by any combination of the following:*

- *A singleton node is both source and sink of the unit graph $g$.*
- *Two series-parallel subgraphs $g_1$ and $g_2$ are joined in series by adding an edge from the sink of $g_1$ to the source of $g_2$, such that $g$'s source is that of $g_1$, and its sink is that of $g_2$.*
- *Series-parallel subgraphs $g_0$, …, $g_{k-1}$ are joined in parallel by adding a source node which links only to the source of each of the subgraphs and a sink node which links only from the sink of each of the subgraphs.*

The particular graphs used in the PAL model are defined as part of the profiling semantics of the models. Figure 5.6 illustrates these and the notation used for defining them:

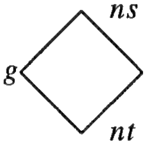
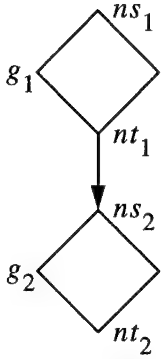
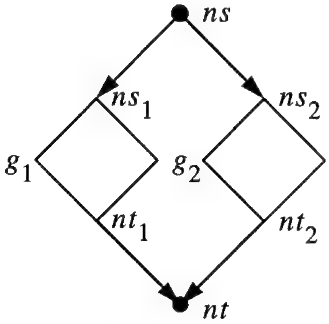
| Graph $g$: | $\mathbf{1}$ | $g_1 \oplus g_2$ | $g_1 \otimes g_2$ |
|---|---|---|---|
| $(ns, nt, NE)$ | $(n, n, \cdot)$<br>unique $n$ | $(ns_1, nt_2,$<br>$(NE_1 \cup NE_2)$<br>$[nt_1 \mapsto [ns_2]])$ | $(ns, nt,$<br>$(NE_1 \cup NE_2)$<br>$[ns \mapsto [ns_1, ns_2]]$<br>$[nt_1 \mapsto [nt]][nt_2 \mapsto [nt]])$<br>unique $ns$ and $nt$ |
|  | |  |  |
| $W(g)$: | 1 | $W(g_1) + W(g_2)$ | $W(g_1) + W(g_2) + 2$ |
| $D(g)$: | 1 | $D(g_1) + D(g_2) + 1$ | $\max(D(g_1), D(g_2)) + 2$ |

Figure 5.6: The definition of combining operators for PAL computation graphs, work, and depth. The informal "unique"-ness side conditions reflect the need to distinguish the nodes in a graph and could be replaced by a counter to generate unique node names.

- A singleton node is represented by $\mathbf{1}$.
- Pairs of graphs are combined in series by $g_1 \oplus g_2$.
- Pairs of graphs are combined in parallel by $g_1 \otimes g_2$. PAL graphs are limited to having binary branching.

The precedence of $\otimes$ is higher than that of $\oplus$. The definitions of these operators are somewhat informal in the use of the "unique"-ness side conditions. These reflect the need to be able to distinguish the nodes in a graph, *i.e.*, that each node is unique. They could be replaced by having a counter used to generate unique node names, but that would clutter each of the profiling semantics.

## 5.3 Simple parallel applicative semantics

This section defines the PAL model based on the basic $\lambda$-calculus of Chapter 4 and its call-by-value semantics of Section 5.1. In this section, the semantics track only the time costs

$$\rho \vdash c \xrightarrow{\text{PAL}} c; \mathbf{1} \qquad \text{(CONST)}$$

$$\rho \vdash \lambda x.e \xrightarrow{\text{PAL}} \mathbf{cl}(\rho,x,e); \mathbf{1} \qquad \text{(LAM)}$$

$$\frac{\rho(x) = v}{\rho \vdash v \xrightarrow{\text{PAL}} v; \mathbf{1}} \qquad \text{(VAR)}$$

$$\frac{\rho \vdash e_1 \xrightarrow{\text{PAL}} \mathbf{cl}(\rho',x,e_3); g_1 \qquad \rho \vdash e_2 \xrightarrow{\text{PAL}} v_2; g_2 \qquad \rho'[x \mapsto v_2] \vdash e_3 \xrightarrow{\text{PAL}} v; g_3}{\rho \vdash e_1\ e_2 \xrightarrow{\text{PAL}} v; (g_1 \otimes g_2) \oplus g_3} \qquad \text{(APP)}$$

$$\frac{\rho \vdash e_1 \xrightarrow{\text{PAL}} c; g_1 \qquad \rho \vdash e_2 \xrightarrow{\text{PAL}} v_2; g_2 \qquad \delta(c, v_2) = v}{\rho \vdash e_1\ e_2 \xrightarrow{\text{PAL}} v; (g_1 \otimes g_2) \oplus g_3} \qquad \text{(APPC)}$$

Figure 5.7: The profiling semantics of the PAL model (without space) using the basic $\lambda$-calculus and the definition of $\delta$ in Figure 5.8.

as described by computation graphs. This serves as the introductory example of a profiling semantics, and Section 5.4 defines a profiling semantics to also define space costs.

The PAL model profiling semantics judgment $\rho \vdash e \xrightarrow{\text{PAL}} v; g$ adds the result computation graph to the operational semantics judgment. Definition 5.4 defines this relation.

**Definition 5.4** (PAL **profiling semantics**) *In the* PAL *model,* in the environment $\rho$, the expression $e$ evaluates to value $v$ with computation graph $g$, *or*

$$\rho \vdash e \xrightarrow{\text{PAL}} v; g,$$

*if it is derivable from the rules of Figure 5.7. Figure 5.8 defines the $\delta$ function for the application of constants.*

Program constants, abstractions, and variables are assumed to evaluate with constant work and depth, *i.e.*, with unit cost **1**. As previously described, an application evaluates the function and argument in parallel. This is followed in serial by evaluation of the function body or constant application, as appropriate.

For each of the numeric constants used here, constant application is assumed to require constant work and depth. However, the semantics allows for a more general cost (the graph returned by $\delta$). *E.g.*, Chapter 9 uses constant functions whose application's work and depth are functions of the argument. Within the context of this model, the constant costs for constant application, or the *uniform cost criterion*, is a reasonable assumption for implementation of these numeric functions on most real machines, assuming we ignore arbitrary precision arithmetic. The semantics could also accommodate the more precise *logarithmic cost criterion* [3].

|  |  | $\delta(c, v)$ |  | if/where |
|---|---|---|---|---|
| $c$ | $v$ | $v'$ | $g'$ |  |
| **add** | $i_1$ | $\mathbf{add}_{i_1}$ | **1** |  |
| $\mathbf{add}_{i_1}$ | $i_2$ | $i_1 + i_2$ | **1** |  |
| **sub** | $i_1$ | $\mathbf{sub}_{i_1}$ | **1** |  |
| $\mathbf{sub}_{i_1}$ | $i_2$ | $i_1 - i_2$ | **1** |  |
| **mul** | $i_1$ | $\mathbf{mul}_{i_1}$ | **1** |  |
| $\mathbf{mul}_{i_1}$ | $i_2$ | $i_1 * i_2$ | **1** |  |
| $\mathbf{div}_i$ | $i'$ | $\lfloor i'/i \rfloor$ | **1** |  |
| **lt** | $i_1$ | $\mathbf{lt}_{i_1}$ | **1** |  |
| $\mathbf{lt}_{i_1}$ | $i_2$ | $\mathbf{cl}(\cdot,x,\lambda y.x)$ | **1** | $i_1 < i_2$ |
| $\mathbf{lt}_{i_1}$ | $i_2$ | $\mathbf{cl}(\cdot,x,\lambda y.y)$ | **1** | $i_1 \geq i_2$ |

Figure 5.8: The $\delta$ function defining constant application for the PAL model.

**Example 5.2** *As a small example of a profiling semantics derivation, observe the evaluation of* $(\lambda x.x)$ $(\lambda y.1)$ 2*. The derivation forms a tree, as Figure 5.9 shows. This is the same as the derivation of Figure 5.4, except that the computation graphs are added. Figure 5.10 shows the overall computation graph.*

## 5.4 Semantics accounting for space

To track space usage we need to model memory more accurately. We must be able to express what data is being shared, so that we do not count its space multiple times. For this we add *stores* which describe a particular state of memory. A store maps a finite set of individual memory *locations* to their contents. Locations are effectively pointers to data structures kept in memory. Together, environments and store provide a level of indirection that allows us to describe sharing.

Introducing stores and locations changes the definitions of values and evaluation somewhat. Following common practice, we do not place constants in the store. For simplicity, we assume that constants are of bounded size[2] (and frequently the same size as a location), the inability to share their storage does not affect space bounds. Thus we distinguish between values in the environment, which are constants or locations, and store-values in the store, which are just closures here. These domains are defined in Figure 5.11, where the set of locations is countably infinite.[3]

[2] This assumption is accurate for fixed-precision arithmetic. But to accurately track the space of the arbitrary-precision numbers allowed in this semantics, this assumption would have to be dropped, and at least some constants would have to be placed in the store like Scheme's "bignums".

[3] This ensures that we have enough locations, and that we can name them (*e.g.*, with the integers).

$$\dfrac{[x \mapsto \mathbf{cl}(\cdot,y,1)](x) = \mathbf{cl}(\cdot,y,1)}{[x \mapsto \mathbf{cl}(\cdot,y,1)] \vdash x \xrightarrow{\text{PAL}} \mathbf{cl}(\cdot,y,1); \mathbf{1}}\,\text{(VAR)} = D_2$$

$$\dfrac{\dfrac{}{\cdot \vdash \lambda x.x \xrightarrow{\text{PAL}} \mathbf{cl}(\cdot,x,x); \mathbf{1}}\,\text{(LAM)} \qquad \dfrac{}{\cdot \vdash \lambda y.1 \xrightarrow{\text{PAL}} \mathbf{cl}(\cdot,y,1); \mathbf{1}}\,\text{(LAM)} \qquad D_2}{\cdot \vdash (\lambda x.x)\ (\lambda y.1) \xrightarrow{\text{PAL}} \mathbf{cl}(\cdot,y,1); (\mathbf{1} \otimes \mathbf{1}) \oplus \mathbf{1}}\,\text{(APP)} = D_1$$

$$\dfrac{D_1 \qquad \dfrac{}{\cdot \vdash 2 \xrightarrow{\text{PAL}} 2; \mathbf{1}}\,\text{(CONST)} \qquad \dfrac{}{[y \mapsto 2] \vdash 1 \xrightarrow{\text{PAL}} 1; \mathbf{1}}\,\text{(CONST)}}{\cdot \vdash (\lambda x.x)\ (\lambda y.1)\ 2 \xrightarrow{\text{PAL}} 1; (((\mathbf{1} \otimes \mathbf{1}) \oplus \mathbf{1}) \otimes \mathbf{1}) \oplus \mathbf{1}}\,\text{(APP)}$$

Figure 5.9: PAL profiling semantics (without space) derivation for Example 5.2. For readability, the derivation tree is broken into three subtrees.

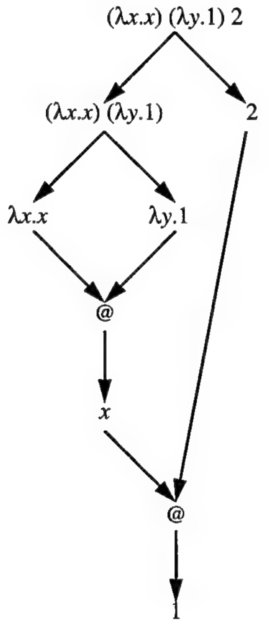


Figure 5.10: PAL computation graph $(((\mathbf{1} \otimes \mathbf{1}) \oplus \mathbf{1}) \otimes \mathbf{1}) \oplus \mathbf{1}$ for Example 5.2. Labeled nodes indicate the expressions evaluated. The "@" nodes each represent the synchronization and application of a function value and its argument.

$$\begin{array}{lcllll}
l & \in & \mathit{Locations} & & & \\
v & \in & \mathit{Values} & ::= & c \mid l & \\
\rho & \in & \mathit{Environments} & = & \mathit{Variables} \overset{\mathit{fin}}{\to} \mathit{Values} & \\
sv & \in & \mathit{StoreValues} & ::= & \mathbf{cl}(\rho,x,e) & \text{closure} \\
\sigma & \in & \mathit{Stores} & = & \mathit{Locations} \overset{\mathit{fin}}{\to} \mathit{StoreValues} & \\
R & \in & \mathit{Roots} & = & \mathit{ValueSets} &
\end{array}$$

Figure 5.11: The definition of PAL run-time domains when tracking space.

The semantics does not need to model garbage collection, but instead, measures the maximum space reachable from a set of *roots* during the evaluation. Our implementation does not include garbage collection either, because that would require extra details obscuring other features. However, Appendix B outlines how garbage collection can be added to ensure that the total space is within a constant factor of the reachable space.

The evaluation relation must now account for these stores and roots. The context of an evaluation is the current environment, store, and roots, and an evaluation results in a value and a new store. Definition 5.5 reflects these changes. Note that the definitions of the two costs—the computation graphs and the reachable space—are independent. We also extend the definition of constant application to use stores, where $\delta(\sigma, c, v)$ now also returns any modifications to the store[4] and also the computation graph of the application (but not the space cost, as we explain later). For the sake of generality, it also returns a computation graph, even though for these constants the graph is always **1**—we take advantage of this generality beginning in Chapter 9. Again, we are primarily interested in evaluations starting at the "top level", where the environment, store, and roots are all empty.

**Definition 5.5 (**PAL **profiling semantics with space)** *In the* PAL *model,* starting with the environment $\rho$, store $\sigma$, and roots $R$, the expression $e$ evaluates to value $v$ and the new store $\sigma'$ with computation graph $g$ and space $s$*, or*

$$\rho, \sigma, R \vdash e \xrightarrow{\mathrm{PAL}} v, \sigma; g, s,$$

*if it is derivable from the rules of Figure 5.12. Figure 5.13 defines the $\delta$ function for the application of constants. Figure 5.14 defines additional functions the reachable space of a computation.*

The space $s$ returned by the semantics represents the maximum reachable space during the computation, assuming a serial evaluation. In particular, in the application rules the store

[4] It returns the modifications to the store, rather than the new store, because that is what the abstract machine uses in Chapter 6.

$$\rho, \sigma, R \vdash c \xrightarrow{\text{PAL}} c, \sigma; \mathbf{1}, S(R, \sigma) \qquad \text{(CONST)}$$

$$\rho, \sigma, R \vdash \lambda x.e \xrightarrow{\text{PAL}} l, \sigma'; \mathbf{1}, S(R \cup \{l\}, \sigma') \quad \text{where } \sigma' = \sigma[l \mapsto \mathbf{cl}(\rho,x,e)],\ l \notin \sigma \qquad \text{(LAM)}$$

$$\frac{\rho(x) = v}{\rho, \sigma, R \vdash v \xrightarrow{\text{PAL}} v, \sigma; \mathbf{1}, S(R \cup \{v\}, \sigma)} \qquad \text{(VAR)}$$

$$\frac{\begin{array}{c}\rho, \sigma, R \cup \rho(FV(e_2)) \vdash e_1 \xrightarrow{\text{PAL}} l, \sigma_1; g_1, s_1 \qquad \rho, \sigma_1, R \cup \{l\} \vdash e_2 \xrightarrow{\text{PAL}} v_2, \sigma_2; g_2, s_2 \\ \sigma_2(l) = \mathbf{cl}(\rho',x,e_3) \qquad \rho'[x \mapsto v_2], \sigma_2, R \vdash e_3 \xrightarrow{\text{PAL}} v, \sigma_3; g_3, s_3\end{array}}{\rho, \sigma, R \vdash e_1\ e_2 \xrightarrow{\text{PAL}} v, \sigma_3; (g_1 \otimes g_2) \oplus g_3, \max(s_1 + 1, s_2 + 1, s_3)} \qquad \text{(APP)}$$

$$\frac{\begin{array}{c}\rho, \sigma, R \cup \rho(FV(e_2)) \vdash e_1 \xrightarrow{\text{PAL}} c, \sigma_1; g_1, s_1 \qquad \rho, \sigma_1, R \vdash e_2 \xrightarrow{\text{PAL}} v_2, \sigma_2; g_2, s_2 \\ \delta(\sigma_2, c, v_2) = v, \sigma_3; g_3\end{array}}{\rho, \sigma, R \vdash e_1\ e_2 \xrightarrow{\text{PAL}} v, \sigma_2 \cup \sigma_3; (g_1 \otimes g_2) \oplus g_3, \max(s_1 + 1, s_2 + 1, S(\sigma, R \cup \{v\}))} \qquad \text{(APPC)}$$

Figure 5.12: The profiling semantics of the PAL model that also accounts for space using the definitions of $\delta$ and $S(\sigma, R)$ in Figures 5.13 and 5.14, respectively.

| | | $\delta(\sigma, c, v)$ | | | |
|---|---|---|---|---|---|
| $c$ | $v$ | $v'$ | $\sigma'$ | $g'$ | if/where |
| **add** | $i_1$ | $\mathbf{add}_{i_1}$, | $\cdot$ | **1** | |
| $\mathbf{add}_{i_1}$ | $i_2$ | $i_1 + i_2$, | $\cdot$ | **1** | |
| **sub** | $i_1$ | $\mathbf{sub}_{i_1}$, | $\cdot$ | **1** | |
| $\mathbf{sub}_{i_1}$ | $i_2$ | $i_1 - i_2$, | $\cdot$ | **1** | |
| **mul** | $i_1$ | $\mathbf{mul}_{i_1}$, | $\cdot$ | **1** | |
| $\mathbf{mul}_{i_1}$ | $i_2$ | $i_1 * i_2$, | $\cdot$ | **1** | |
| $\mathbf{div}_i$ | $i'$ | $\lfloor i'/i \rfloor$, | $\cdot$ | **1** | |
| **lt** | $i_1$ | $\mathbf{lt}_{i_1}$, | $\cdot$ | **1** | |
| $\mathbf{lt}_{i_1}$ | $i_2$ | $l$, | $[l \mapsto \mathbf{cl}(\cdot,x,\lambda y.x)]$ | **1** | $i_1 < i_2,\ l \notin \sigma$ |
| $\mathbf{lt}_{i_1}$ | $i_2$ | $l$, | $[l \mapsto \mathbf{cl}(\cdot,x,\lambda y.y)]$ | **1** | $i_1 \geq i_2,\ l \notin \sigma$ |

Figure 5.13: The $\delta$ function defining constant application for the PAL model that also accounts for space.

$$\begin{array}{rcl} S(R,\sigma) & = & \sum_{l\in L} |FV(e) - \{x\}| + 2 \;\; \text{where } \sigma(l) = \mathbf{cl}(-,x,e) \\ & & \text{where } L = \bigcup_{l\in R} \mathit{locs}(l,\sigma) \end{array}$$

$$\begin{array}{rcl} \mathit{locs}(c, \quad\quad \sigma) & = & \{\} \\ \mathit{locs}(l, \quad\quad \sigma) & = & \{l\} \cup \mathit{locs}(\sigma(l),\sigma) \\ & & \\ \mathit{locs}(\mathbf{cl}(\rho,x,e),\, \sigma) & = & \bigcup_{l\in L} \mathit{locs}(l,\sigma) \\ & & \text{where } L = \rho(FV(e) - \{x\}) \end{array}$$

Figure 5.14: Semantics functions used for defining reachable space in the PAL model.

is threaded through the sub-evaluations. Since it reflects a serial evaluation, rather than the PAL model's parallel evaluation, its definition may seem out of place in the same semantics, but we present them together for convenience. Chapters 6 and 7 relate this serial space to the space required during parallel evaluation, which depends on the number of processors used. The semantics accounts for the reachable space by tracking all the values that might be needed in the continuation. These are kept as a set of labels $R$ into the store. For example, in a function application $e_1\ e_2$ when executing $e_1$, the semantics adds the labels for the free variables in $e_2$ to the current set of labels. Given a set of root labels, the space required by the data is measured by finding all the locations reachable from the root locations $R$, and summing the space for each object stored at these labels (see Figure 5.14). Space is measured only at the leaves of the evaluation tree (in the rules CONST, LAM, VAR, and APPC). The addition of 1 to the space in the two application rules represents the space needed for control information. *I.e.*, during the execution of the function, we need to store a pointer to $e_2$, and during the execution of the argument, we need to store $l$.

**Example 5.3** *We return to our previous example program, and show the profiling semantics derivation for the expression* $(\lambda x.x)$ $(\lambda y.1)$ 2. *Figure 5.15 shows the derivation tree, which mirrors that of Figure 5.9.*

## Observations

The $\delta$ function doesn't return a space cost like the profiling semantics judgments. The function is like a special base case of the semantics since it does not use the semantics recursively, and the semantics measures space at the base cases with the function $S(R,\sigma)$. The semantics could pass the roots to $\delta$ and calculate the space there, rather than in APPC. But since in all of our constant functions, the space of the application is bounded by that of the result value, we use a simpler notation.

$$\dfrac{\sigma'(x) = l'}{[x \mapsto l'], \sigma', \{\} \vdash x \xrightarrow{\text{PAL}} l', \sigma'; \mathbf{1}, 2}\,(\text{VAR}) = D_2$$

$$\dfrac{\dfrac{}{\cdot, \cdot, \{\} \vdash \lambda x.x \xrightarrow{\text{PAL}} l, \sigma; \mathbf{1}, 3}\,(\text{LAM}) \qquad \dfrac{}{\cdot, \sigma, \{l\} \vdash \lambda y.1 \xrightarrow{\text{PAL}} l', \sigma'; \mathbf{1}, 6}\,(\text{LAM}) \qquad D_2}{\cdot, \sigma', \{\} \vdash (\lambda x.x)\ (\lambda y.1) \xrightarrow{\text{PAL}} l', \sigma'; (\mathbf{1} \otimes \mathbf{1}) \oplus \mathbf{1}, 7}\,(\text{APP}) = D_1$$

$$\dfrac{D_1 \qquad \dfrac{}{\cdot, \sigma', \{l'\} \vdash 2 \xrightarrow{\text{PAL}} 2, \sigma'; \mathbf{1}, 2}\,(\text{CONST}) \qquad \dfrac{}{[y \mapsto 2], \sigma', \{\} \vdash 1 \xrightarrow{\text{PAL}} 1, \sigma'; \mathbf{1}, 0}\,(\text{CONST})}{\cdot, \cdot, \{\} \vdash (\lambda x.x)\ (\lambda y.1)\ 2 \xrightarrow{\text{PAL}} 1, \sigma'; (((\mathbf{1} \otimes \mathbf{1}) \oplus \mathbf{1}) \otimes \mathbf{1}) \oplus \mathbf{1}, 8}\,(\text{APP})$$

$$\text{where} \quad \begin{array}{lcl} \sigma & = & [l \mapsto \mathbf{cl}(\cdot, x, x)] \\ \sigma' & = & \sigma[l' \mapsto \mathbf{cl}(\cdot, y, 1)] \end{array}$$

Figure 5.15: PAL profiling semantics derivation for Example 5.3. For readability, the derivation tree is broken into three subtrees.

The resulting store of an evaluation $\rho, \sigma \vdash e \xrightarrow{\text{PAL}} v, \sigma'; g, s$, can be factored into the original store plus some new bindings:

$$\sigma' = \sigma \cup \sigma_{new},$$

where $\sigma_{new}$ refers only to new locations.[5] This observation is used in Lemma 5.1. We could have defined the semantics to return $\sigma_{new}$ instead of $\sigma'$, like the definition of $\delta$, but the style used is standard.

## 5.5 Equivalence of λ-calculus and extended λ-calculus

This section shows that a PAL model based on the extended λ-calculus (the PAL' model) shares the same asymptotic bounds as the previous PAL model. To prove this, we give a syntactic translation from the PAL' model to the PAL model, and prove that the translation introduces only constant overheads in time and space. This shows that is it sufficient for us to use only the core of the λ-calculus and that any asymptotic cost bounds we prove later for the PAL model also hold for the richer PAL' model. The converse holds almost trivially. Similar results would also hold for the addition of many other standard features of programming languages, including general data-structures, pattern matching as in ML or Haskell, and loop constructs.

Section 5.5.1 defines the semantics of the PAL' model. Then Section 5.5.2 defines the translation between the models and proves their asymptotic equivalence.

[5] In general, the semantics could reuse old locations, but that doesn't make sense for the constant functions of the PAL model.

$$\begin{array}{lcllll}
l & \in & \mathit{Locations} & & & \\
v & \in & \mathit{Values} & ::= & c \mid l & \\
\rho & \in & \mathit{Environments} & = & \mathit{Variables} \overset{fin}{\rightarrow} \mathit{Values} & \\
sv & \in & \mathit{StoreValues} & ::= & \mathbf{cl}(\rho,x,y,e) \mid & \text{recursive closure} \\
 & & & & \langle v_1,v_2\rangle & \text{pair} \\
\sigma & \in & \mathit{Stores} & = & \mathit{Locations} \overset{fin}{\rightarrow} \mathit{StoreValues} & \\
R & \in & \mathit{Roots} & = & \mathit{ValueSets} &
\end{array}$$

Figure 5.16: The definition of PAL' run-time domains.

### 5.5.1 Semantics for the extended $\lambda$-calculus

The store-values of the PAL' model are somewhat different for those of the basic PAL model, although they still include constants and closures. Recall, however, that a different set of constants is used for the extended language. Also, to accommodate the explicit recursion of **letrec**, closures are now named—this allows us to avoid using recursive environments, which allows simpler proofs. In addition, the store-values also include pairs. So, values are defined as in Figure 5.16, where in the closure, $x$ is its name, and $y$ is its bound variable.

The semantic rules for defining evaluation and its costs are similar to before, and given by Definition 5.6. The differences from the PAL model are as follows:

- The LAM rule results in a closure with the special name _, a dummy variable.

- The appropriate PAIR, LETREC, and IF rules are added:

  The PAIR rule creates a new pair in the store. It evaluates the subexpressions in parallel, and then creates the pair puts into a new location in the store.

  The LETREC rule creates a new recursive closure in the store for the evaluation of the body $e_2$. Note that the closure's environment does not include this closure.

  The IF rules evaluate in serial the test expression $e_1$ and the appropriate branch $e_2$ or $e_3$. The live data during the condition test expression includes the data that may be used in either branch.

- The APP rule uses a potentially recursive closure, unrolling the recursion in the environment once per application.

- The $\delta$ rules use the new set of constants. In particular, the binary constant functions are now *uncurried*, *i.e.*, they take pairs of arguments, rather than arguments one at a time.

**Definition 5.6** **(PAL' profiling semantics)** *In the* PAL*' model,* starting with the environment $\rho$, store $\sigma$, and roots $R$, the expression $e$ evaluates to value $v$ and the new store $\sigma'$ with computation graph $g$ and space $s$, *or*

$$\rho, \sigma, R \vdash e \xrightarrow{\text{PAL}'} x, \sigma'; g, s,$$

*if it derivable from the rules of Figure 5.17. Figure 5.18 defines the $\delta$ function for the application of constants. Figure 5.19 defines additional functions for the reachable space of a computation.*

### 5.5.2 Equivalence of the PAL and PAL' models

The PAL and PAL' models are essentially equivalent in that they can simulate each other with only a constant factor of overhead:

- The PAL' model is an extension of the PAL model, except for the difference between curried and uncurried constant functions. The PAL' simulation of the PAL introduces overhead for pairs of arguments to these uncurried functions, but these correspond to the closures created in the PAL curried applications. Thus this does not introduce more than a constant factor of overhead.

- The PAL simulation of the PAL' model requires translating the extensions of the PAL' into PAL. The remainder of this section gives such a translation $T_{\text{PAL}}[\![]\!]$, proves that $T_{\text{PAL}}[\![]\!]$ is correct, and proves that $T_{\text{PAL}}[\![]\!]$ introduces only a constant factor of overhead.

Definition 5.7 defines a translation from the semantic domains of the PAL' model to those of the PAL model, and Theorem 5.1 shows that the translated evaluation derivation is correct and incurs at most a constant factor of overhead. The PAL derivation uses an initial environment and store defining the PAL' model's extra constants. Then for the most part, the proof follows directly from induction on the structure of the PAL' derivation, although extra complications arise for the simulation of explicit recursion.

**Definition 5.7** *Figure 5.20 shows the translation $T_{\text{PAL}}[\![]\!]$ of* PAL*' expressions, values, and store-values to those of the* PAL *model. The translations of environments and stores are defined point-wise on the values and store-values in their ranges, respectively, except that the translation of the environment omits any bindings for the dummy variable* _.[6] *The translation of root sets of values is defined point-wise on the contents.*

The translation uses standard encodings, but introduces three novel differences:

[6] This omission corresponds to the lack of a binding for the dummy variable _ in the translation of a non-recursive PAL' closure.

$$\rho,\sigma,R \vdash c \xrightarrow{\mathrm{PAL}'} c,\sigma;\mathbf{1},S(R,\sigma) \qquad \text{(CONST)}$$

$$\rho,\sigma,R \vdash \lambda x.e \xrightarrow{\mathrm{PAL}'} l,\sigma';\mathbf{1},S(R\cup\{l\},\sigma') \quad \text{where } \sigma' = \sigma[l \mapsto \mathbf{cl}(\rho,\_,x,e)],\ l \notin \sigma \qquad \text{(LAM)}$$

$$\frac{\rho(x) = v}{\rho,\sigma,R \vdash v \xrightarrow{\mathrm{PAL}'} v,\sigma;\mathbf{1},S(R\cup\{\rho(x)\},\sigma)} \qquad \text{(VAR)}$$

$$\frac{\begin{array}{c}\rho,\sigma,R\cup\rho(FV(e_2)) \vdash e_1 \xrightarrow{\mathrm{PAL}'} l,\sigma_1;g_1,s_1 \qquad \rho,\sigma_1,R\cup\{l\} \vdash e_2 \xrightarrow{\mathrm{PAL}'} v_2,\sigma_2;g_2,s_2\\ \sigma_2(l) = \mathbf{cl}(\rho',x,y,e_3) \qquad \rho'[x\mapsto l][y\mapsto v_2],\sigma_2,R \vdash e_3 \xrightarrow{\mathrm{PAL}'} v_3,\sigma_3;g_3,s_3\end{array}}{\rho,\sigma \vdash e_1\ e_2 \xrightarrow{\mathrm{PAL}'} v_3,\sigma_3;(g_1\otimes g_2)\oplus g_3,\mathrm{max}(s_1+1,s_2+1,s_3)} \qquad \text{(APP)}$$

$$\frac{\begin{array}{c}\rho,\sigma,R\cup\rho(FV(e_2)) \vdash e_1 \xrightarrow{\mathrm{PAL}'} c,\sigma_1;g_1,s_1 \qquad \rho,\sigma_1,R \vdash e_2 \xrightarrow{\mathrm{PAL}'} v_2,\sigma_2;g_2,s_2\\ \delta(\sigma_2,c,v_2) = v_3,\sigma_3;g_3\end{array}}{\rho,\sigma,R \vdash e_1\ e_2 \xrightarrow{\mathrm{PAL}'} v_3,\sigma_2\cup\sigma_3;(g_1\otimes g_2)\oplus g_3,\mathrm{max}(s_1+1,s_2+1,S(R\cup\{v\},\sigma_3))} \qquad \text{(APPC)}$$

$$\frac{\begin{array}{c}\rho,\sigma,R\cup\rho(FV(e_2)) \vdash e_1 \xrightarrow{\mathrm{PAL}'} v_1,\sigma_1;g_1,s_1\\ \rho,\sigma_1,R\cup\{v_1\} \vdash e_2 \xrightarrow{\mathrm{PAL}'} v_2,\sigma_2;g_2,s_2\end{array}}{\rho,\sigma,R \vdash (e_1,e_2) \xrightarrow{\mathrm{PAL}'} l,\sigma_2[l\mapsto\langle v_1,v_2\rangle];(g_1\otimes g_2)\oplus\mathbf{1},\mathrm{max}(s_1+1,s_2)} \quad \text{where } l\notin\sigma \qquad \text{(PAIR)}$$

$$\frac{\rho[x\mapsto l],\sigma[l\mapsto\mathbf{cl}(\rho,x,y,e_1)],R \vdash e_2 \xrightarrow{\mathrm{PAL}'} v,\sigma';g_2,s}{\rho,\sigma,R \vdash \mathbf{letrec}\ x\ y = e_1\ \mathbf{in}\ e_2 \xrightarrow{\mathrm{PAL}'} v,\sigma';1\oplus g_2,s+1} \quad \text{where } l\notin\sigma \qquad \text{(LETREC)}$$

$$\frac{\begin{array}{c}\rho,\sigma,R\cup\rho(FV(e_2))\cup\rho(FV(e_3)) \vdash e_1 \xrightarrow{\mathrm{PAL}'} \mathbf{true},\sigma_1;g_1,s_1\\ \rho,\sigma_1,R \vdash e_2 \xrightarrow{\mathrm{PAL}'} v_2,\sigma_2;g_2,s_2\end{array}}{\rho,\sigma,R \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \xrightarrow{\mathrm{PAL}'} v_2,\sigma_2;\mathbf{1}\oplus g_1\oplus g_2,\mathrm{max}(s_1+1,s_2)} \qquad \text{(IF-TRUE)}$$

$$\frac{\begin{array}{c}\rho,\sigma,R\cup\rho(FV(e_2))\cup\rho(FV(e_3)) \vdash e_1 \xrightarrow{\mathrm{PAL}'} \mathbf{false},\sigma_1;g_1,s_1\\ \rho,\sigma_1,R \vdash e_3 \xrightarrow{\mathrm{PAL}'} v_3,\sigma_3;g_3,s_3\end{array}}{\rho,\sigma,R \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \xrightarrow{\mathrm{PAL}'} v_3,\sigma_3;\mathbf{1}\oplus g_1\oplus g_3,\mathrm{max}(s_1+1,s_3)} \qquad \text{(IF-FALSE)}$$

Figure 5.17: The profiling semantics of the PAL' model using the extended $\lambda$-calculus and the definitions of $\delta$ and $S(\sigma, R)$ in Figures 5.18 and 5.19, respectively.

| | | $\delta(\sigma, c, v)$ | | | |
|---|---|---|---|---|---|
| $c$ | $v$ | $v'$ | $\sigma'$ | $g'$ | if/where |
| **add** | $l$ | $i_1 + i_2$ | $\cdot$ | **1** | $\sigma(l) = \langle i_1, i_2 \rangle$ |
| **sub** | $l$ | $i_1 - i_2$ | $\cdot$ | **1** | $\sigma(l) = \langle i_1, i_2 \rangle$ |
| **mul** | $l$ | $i_1 * i_2$ | $\cdot$ | **1** | $\sigma(l) = \langle i_1, i_2 \rangle$ |
| $\mathbf{div}_i$ | $i'$ | $\lfloor i'/i \rfloor,$ | $\cdot$ | **1** | |
| **lt** | $l$ | **true** | $\cdot$ | **1** | $\sigma(l) = \langle i_1, i_2 \rangle,\ i_1 < i_2$ |
| **lt** | $l$ | **false** | $\cdot$ | **1** | $\sigma(l) = \langle i_1, i_2 \rangle,\ i_1 \geq i_2$ |
| **eq** | $l$ | **true** | $\cdot$ | **1** | $\sigma(l) = \langle i_1, i_2 \rangle,\ i_1 = i_2$ |
| **eq** | $l$ | **false** | $\cdot$ | **1** | $\sigma(l) = \langle i_1, i_2 \rangle,\ i_1 \neq i_2$ |
| **gt** | $l$ | **true** | $\cdot$ | **1** | $\sigma(l) = \langle i_1, i_2 \rangle,\ i_1 > i_2$ |
| **gt** | $l$ | **false** | $\cdot$ | **1** | $\sigma(l) = \langle i_1, i_2 \rangle,\ i_1 \leq i_2$ |
| **not** | $b$ | **false** | $\cdot$ | **1** | $b = \mathbf{true}$ |
| **not** | $b$ | **true** | $\cdot$ | **1** | $b = \mathbf{false}$ |
| **and** | $l$ | $b_2$ | $\cdot$ | **1** | $\sigma(l) = \langle \mathbf{true}, b_2 \rangle$ |
| **and** | $l$ | **false** | $\cdot$ | **1** | $\sigma(l) = \langle \mathbf{false}, b_2 \rangle$ |
| **or** | $l$ | **true** | $\cdot$ | **1** | $\sigma(l) = \langle \mathbf{true}, b_2 \rangle$ |
| **or** | $l$ | $b_2$ | $\cdot$ | **1** | $\sigma(l) = \langle \mathbf{false}, b_2 \rangle$ |
| **fst** | $l$ | $v_1$ | $\cdot$ | **1** | $\sigma(l) = \langle v_1, v_2 \rangle$ |
| **snd** | $l$ | $v_2$ | $\cdot$ | **1** | $\sigma(l) = \langle v_1, v_2 \rangle$ |

Figure 5.18: The $\delta$ function defining constant application for the PAL' model with the extended $\lambda$-calculus.

$$
\begin{array}{lcl}
S(R, \sigma) & = & \sum_{l \in L} \begin{cases} |FV(e) - \{x, y\}| + 2 & \text{if } \sigma(l) = \mathbf{cl}(-, x, y, e) \\ 2 & \text{if } \sigma(l) = \langle -, - \rangle \end{cases} \\
 & & \text{where } L = \bigcup_{l \in R} \mathit{locs}(l, \sigma)
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{locs}(c, \qquad \sigma) & = & \{\} \\
\mathit{locs}(l, \qquad \sigma) & = & \{l\} \cup \mathit{locs}(\sigma(l), \sigma) \\
 & & \\
\mathit{locs}(\mathbf{cl}(\rho, x, y, e), \sigma) & = & \bigcup_{l \in L} \mathit{locs}(l, \sigma) \\
 & & \text{where } L = \rho(FV(e) - \{x, y\}) \\
\mathit{locs}(\langle v_1, v_2 \rangle, \qquad \sigma) & = & \mathit{locs}(v_1, \sigma) \cup \mathit{locs}(v_2, \sigma)
\end{array}
$$

Figure 5.19: Semantics functions used for defining reachable space in the PAL' model.

Expressions:

$$\begin{array}{rcl}
\mathcal{T}_{\mathrm{PAL}}\llbracket i\rrbracket & = & i \\
\mathcal{T}_{\mathrm{PAL}}\llbracket \mathbf{div}_i\rrbracket & = & \mathbf{div}_i \\
\mathcal{T}_{\mathrm{PAL}}\llbracket c\rrbracket & = & x_c,\text{ if } c \notin \{i, \mathbf{div}_i\} \\
\mathcal{T}_{\mathrm{PAL}}\llbracket x\rrbracket & = & x \\
\mathcal{T}_{\mathrm{PAL}}\llbracket \lambda x.e\rrbracket & = & \lambda x.\mathcal{T}_{\mathrm{PAL}}\llbracket e\rrbracket \\
\mathcal{T}_{\mathrm{PAL}}\llbracket e_1\ e_2\rrbracket & = & \mathcal{T}_{\mathrm{PAL}}\llbracket e_1\rrbracket\ \mathcal{T}_{\mathrm{PAL}}\llbracket e_2\rrbracket \\
\mathcal{T}_{\mathrm{PAL}}\llbracket \mathbf{letrec}\ x\ y = e_1\ \mathbf{in}\ e_2\rrbracket & = & (\lambda x.\mathcal{T}_{\mathrm{PAL}}\llbracket e_2\rrbracket)\ (x_Y\ (\lambda x.\lambda y.\mathcal{T}_{\mathrm{PAL}}\llbracket e_1\rrbracket)) \\
\mathcal{T}_{\mathrm{PAL}}\llbracket (e_1,e_2)\rrbracket & = & x_P\ \mathcal{T}_{\mathrm{PAL}}\llbracket e_1\rrbracket\ \mathcal{T}_{\mathrm{PAL}}\llbracket e_2\rrbracket \\
\mathcal{T}_{\mathrm{PAL}}\llbracket \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3\rrbracket & = & \mathcal{T}_{\mathrm{PAL}}\llbracket e_1\rrbracket\ (\lambda x.\mathcal{T}_{\mathrm{PAL}}\llbracket e_2\rrbracket)\ (\lambda x.\mathcal{T}_{\mathrm{PAL}}\llbracket e_3\rrbracket)\ 0
\end{array}$$

Values:

$$\begin{array}{ll}
\mathcal{T}_{\mathrm{PAL}}\llbracket i\rrbracket_{\sigma'}\llbracket i\rrbracket_{\sigma} & \\
\mathcal{T}_{\mathrm{PAL}}\llbracket c\rrbracket_{\sigma'}\llbracket l_c\rrbracket_{\sigma} & \text{if } l_c \in \sigma \\
\mathcal{T}_{\mathrm{PAL}}\llbracket l'\rrbracket_{\sigma'}\llbracket l\rrbracket_{\sigma} & \text{if } \mathcal{T}_{\mathrm{PAL}}\llbracket l'(\sigma')\rrbracket_{\sigma'}\llbracket l(\sigma)\rrbracket_{\sigma}
\end{array}$$

Store-Values:

$$\begin{array}{ll}
\mathcal{T}_{\mathrm{PAL}}\llbracket \mathbf{cl}(\rho',\_,x,e)\rrbracket_{\sigma'} & \text{if } \mathcal{T}_{\mathrm{PAL}}\llbracket \rho'\rrbracket_{\sigma'}\llbracket \rho\rrbracket_{\sigma} \\
\quad \llbracket \mathbf{cl}(\rho,x,\mathcal{T}_{\mathrm{PAL}}\llbracket e\rrbracket)\rrbracket_{\sigma} & \\
\mathcal{T}_{\mathrm{PAL}}\llbracket \mathbf{cl}(\rho',x,y,e)\rrbracket_{\sigma'} & \text{if } x \neq \_,\ \mathcal{T}_{\mathrm{PAL}}\llbracket \rho'\rrbracket_{\sigma'}\llbracket \rho\rrbracket_{\sigma}, \\
\quad \llbracket \mathbf{cl}(\rho[x \mapsto l_{Y1}],y,\mathcal{T}_{\mathrm{PAL}}\llbracket e\rrbracket)\rrbracket_{\sigma} & \quad \sigma(l_{Y1}) = \mathbf{cl}(\rho[y' \mapsto l_{Y2}],z',y'\ y'\ z'), \\
 & \quad \sigma(l_{Y2}) = \mathbf{cl}(\rho_Y,y',x'\ (\lambda z'.y'\ y'\ z')), \\
 & \quad \rho_Y = \rho[x' \mapsto \mathbf{cl}(\rho,x,\lambda y.\mathcal{T}_{\mathrm{PAL}}\llbracket e\rrbracket)] \\
\mathcal{T}_{\mathrm{PAL}}\llbracket \mathbf{cl}(\rho',x,y,e)\rrbracket_{\sigma'} & \text{if } x \neq \_,\ \mathcal{T}_{\mathrm{PAL}}\llbracket \rho'\rrbracket_{\sigma'}\llbracket \rho\rrbracket_{\sigma}, \\
\quad \llbracket \mathbf{cl}(\rho[y' \mapsto l_{Y2}],z',y'\ y'\ z')\rrbracket_{\sigma} & \quad \sigma(l_{Y2}) = \mathbf{cl}(\rho_Y,y',x'\ (\lambda z'.y'\ y'\ z')), \\
 & \quad \rho_Y = \rho[x' \mapsto \mathbf{cl}(\rho,x,\lambda y.\mathcal{T}_{\mathrm{PAL}}\llbracket e\rrbracket)] \\
\mathcal{T}_{\mathrm{PAL}}\llbracket \langle v_1',v_2'\rangle\rrbracket_{\sigma'} & \text{if } \mathcal{T}_{\mathrm{PAL}}\llbracket v_1'\rrbracket_{\sigma'}\llbracket v_1\rrbracket_{\sigma},\ \mathcal{T}_{\mathrm{PAL}}\llbracket v_2'\rrbracket_{\sigma'}\llbracket v_2\rrbracket_{\sigma} \\
\quad \llbracket \mathbf{cl}([x \mapsto v_1][y \mapsto v_2],z,z\ x\ y)\rrbracket_{\sigma} &
\end{array}$$

Figure 5.20: Translation $\mathcal{T}_{\mathrm{PAL}}\llbracket\rrbracket$ from the PAL' model expressions, values, and store-values to those of the PAL model. The translation on expressions is presented as a function. Any new variables ($x_c$, $x_P$, $x_Y$, or those primed) are assumed to be distinct from the free variables of the expression or closure being translated. The variables $x_c$, $x_P$, and $x_Y$ are defined in the initial environment (Figure 5.21).

| | $\rho_I$ | |
|---|---|---|
| $x_c$ | $\mapsto$ | $l_c$, for each $c \notin \{i, \mathbf{div}_i\}$ |
| $x_P$ | $\mapsto$ | $l_P$ |
| $x_Y$ | $\mapsto$ | $l_Y$ |

| | $\sigma_I$ | |
|---|---|---|
| $l_c$ | $\mapsto$ | $\mathbf{cl}(\cdot,x,e_c)$, for each $c \in \{\mathbf{add}, \mathbf{sub}, \mathbf{mul}, \mathbf{lt}, \mathbf{eq}, \mathbf{gt},$<br>$\mathbf{true}, \mathbf{false}, \mathbf{not}, \mathbf{and}, \mathbf{or}, \mathbf{fst}, \mathbf{snd}\}$ |
| $l_P$ | $\mapsto$ | $\mathbf{cl}(\cdot,x,e_P)$ |
| $l_Y$ | $\mapsto$ | $\mathbf{cl}(\cdot,x,e_Y)$ |
| | | |
| $e_c$ | $=$ | $\mathit{uncurry}\llbracket c \rrbracket$, for each $c \in \{\mathbf{add}, \mathbf{sub}, \mathbf{mul}, \mathbf{lt}\}$ |
| $e_{\mathbf{eq}}$ | $=$ | $\mathit{uncurry}\llbracket \lambda x.\lambda y.\mathcal{T}_{\mathrm{PAL}}\llbracket \mathbf{not}\ (\mathbf{or}\ (\mathbf{lt}\ x\ y)\ (\mathbf{lt}\ x\ y)) \rrbracket \rrbracket$ |
| $e_{\mathbf{gt}}$ | $=$ | $\mathit{uncurry}\llbracket \lambda x.\lambda y.\mathcal{T}_{\mathrm{PAL}}\llbracket \mathbf{lt}\ y\ x \rrbracket \rrbracket$ |
| $e_{\mathbf{true}}$ | $=$ | $\lambda y.x$ |
| $e_{\mathbf{false}}$ | $=$ | $\lambda y.y$ |
| $e_{\mathbf{not}}$ | $=$ | $\mathcal{T}_{\mathrm{PAL}}\llbracket \mathbf{if}\ x\ \mathbf{then\ false\ else\ true} \rrbracket$ |
| $e_{\mathbf{and}}$ | $=$ | $\mathit{uncurry}\llbracket \lambda x.\lambda y.\mathcal{T}_{\mathrm{PAL}}\llbracket \mathbf{if}\ x\ \mathbf{then}\ y\ \mathbf{else\ false} \rrbracket \rrbracket$ |
| $e_{\mathbf{or}}$ | $=$ | $\mathit{uncurry}\llbracket \lambda x.\lambda y.\mathcal{T}_{\mathrm{PAL}}\llbracket \mathbf{if}\ x\ \mathbf{then\ true\ else}\ y \rrbracket \rrbracket$ |
| $e_{\mathbf{fst}}$ | $=$ | $x\ (\lambda y.\lambda z.y)$ |
| $e_{\mathbf{snd}}$ | $=$ | $x\ (\lambda y.\lambda z.z)$ |
| $e_P$ | $=$ | $\lambda y.\lambda z.z\ x\ y$ |
| $e_Y$ | $=$ | $(\lambda y.x\ (\lambda z.y\ y\ z))\ (\lambda y'.x\ (\lambda z.y\ y\ z))$ |
| | | |
| where $\mathit{uncurry}\llbracket e \rrbracket = e\ (\mathcal{T}_{\mathrm{PAL}}\llbracket \mathbf{fst} \rrbracket\ x)\ (\mathcal{T}_{\mathrm{PAL}}\llbracket \mathbf{snd} \rrbracket\ x)$ | | |

Figure 5.21: Initial environment $\rho_I$ and store $\sigma_I$ of the PAL model when translating from the PAL' model with $\mathcal{T}_{\mathrm{PAL}}\llbracket\rrbracket$. Note the correspondence of $x_P$ and $x_Y$ to the translations of store-values for pairs and recursive closures, respectively.

$$\frac{\rho'[x \mapsto l], \sigma'[l \mapsto \mathbf{cl}(\rho',x,y,e_1)] \vdash e_2 \xrightarrow{\text{PAL}'} v', \sigma'_{new}; g_2}{\rho', \sigma' \vdash \mathbf{letrec}\ x\ y = e_1\ \mathbf{in}\ e_2 \xrightarrow{\text{PAL}'} v', \sigma'_{new}; \mathbf{1} \oplus g_2}\ \text{(LETREC)}$$

Figure 5.22: PAL' derivation with LETREC, excluding space costs.

- It translates **letrec** expressions and recursive closures using the call-by-value version of the *Y-combinator*, a standard way of encoding recursion. The basic property of the $Y$-combinator is that, given a function $F$ of the appropriate form, the application $Y\ F$ behaves the same as $F\ (Y\ F)$, *i.e.*, $Y\ F$ is the fixed point of $F$.

  This behavioral equivalence complicates the translation of values because of the difference between the APP rules of the PAL and PAL' models. Let's examine what happens when we apply a translated PAL' recursive function in the PAL model. Figure 5.22 shows the derivation for a PAL' **letrec**, and Figure 5.23 shows the derivation for its PAL translation.

  Observe that the translation of **letrec** introduces an expression of the form $Y\ F$. For this analogy, $Y$ corresponds to $x_Y$, and $F$ to $\lambda x.\lambda y.\mathcal{T}_{\text{PAL}}[\![e_1]\!]$. Thus the resulting value of this application, which is eventually stored at location $l_6$, corresponds to the value of $Y\ F$.

  Now assume that the recursive function $x$ is applied within the **letrec** body $e_2$. Initially, $x$ is bound to $l_6$, or intuitively, $Y\ F$. During the application, the body $\mathcal{T}_{\text{PAL}}[\![e_1]\!]$ of the recursive function is evaluated in the environment where the $x$ is bound to $l_5$, or intuitively, $F\ (Y\ F)$. Thus, during the PAL evaluation, $x$ is bound to two different, but equivalent, values. However, in the PAL' model, $x$ is only bound to the same value, the location containing the recursive closure.

  While the two PAL values behave similarly, they are not identical, and a straightforward translation does not accommodate the difference. *Thus we provide a translation which is not a function, but a relation, and which maps* PAL*' recursive closures to both of these* PAL *closures.*[7] Observe that the closures stored in $l_6$ and $l_5$ in Figure 5.23 correspond to the second and third lines, respectively, of the definition of $\mathcal{T}_{\text{PAL}}[\![]\!]$ on store-values.

  Furthermore, by comparing contents, not names, of locations, this relation also accommodates the fact that each unrolling of a recursive function generates a separate location containing a copy of the $F\ (Y\ F)$-like closure. This extra generality allows a simpler proof of model equivalence than otherwise possible.

- It translates constants to variables, which are initially bound to the appropriate function closures of Figure 5.21. The exceptions are $i$ and $\mathbf{div}_i$ as they are the same in both models.

[7] The translation on expressions is still a function.

$$\frac{}{\rho_3, \sigma_5 \vdash \lambda y.T_{\mathrm{PAL}}[\![e_1]\!] \xrightarrow{\mathrm{PAL}} l_6, \sigma_6; \mathbf{1}}\,(\mathrm{LAM}) = D_4$$

$$\frac{\dfrac{\rho_2(x') = l_2}{\rho_2, \sigma_4 \vdash x' \xrightarrow{\mathrm{PAL}} l_2, \sigma_4; \mathbf{1}}\,(\mathrm{VAR}) \qquad \dfrac{}{\rho_2, \sigma_4 \vdash \lambda z'.x'\ x'\ z' \xrightarrow{\mathrm{PAL}} l_5, \sigma_5; \mathbf{1}}\,(\mathrm{LAM}) \qquad D_4}{\rho_2, \sigma_4 \vdash e' \xrightarrow{\mathrm{PAL}} l_6, \sigma_6; g'}\,(\mathrm{APP}) = D_3$$

$$\frac{\dfrac{}{\rho_1, \sigma_2 \vdash \lambda y'.e' \xrightarrow{\mathrm{PAL}} l_3, \sigma_3; \mathbf{1}}\,(\mathrm{LAM}) \qquad \dfrac{}{\rho_1, \sigma_3 \vdash \lambda y'.e' \xrightarrow{\mathrm{PAL}} l_4, \sigma_4; \mathbf{1}}\,(\mathrm{LAM}) \qquad D_3}{\rho_1, \sigma_2 \vdash (\lambda y'.e')\ (\lambda y'.e') \xrightarrow{\mathrm{PAL}} l_6, \sigma_6; g'}\,(\mathrm{APP}) = D_2$$

$$\frac{\dfrac{\sigma_0(x_Y) = l_Y}{\rho_0, \sigma_1 \vdash x_Y \xrightarrow{\mathrm{PAL}} l_Y, \sigma_0; \mathbf{1}}\,(\mathrm{VAR}) \qquad \dfrac{}{\rho_0, \sigma_1 \vdash \lambda x.\lambda y.T_{\mathrm{PAL}}[\![e_1]\!] \xrightarrow{\mathrm{PAL}} l_2, \sigma_2; \mathbf{1}}\,(\mathrm{LAM}) \qquad D_2}{\rho_0, \sigma_1 \vdash x_Y\ (\lambda x.\lambda y.T_{\mathrm{PAL}}[\![e_1]\!]) \xrightarrow{\mathrm{PAL}} l_6, \sigma_6; g''}\,(\mathrm{APP}) = D_1$$

$$\frac{\dfrac{}{\rho_I \cup \rho, \sigma_0 \vdash \lambda x.T_{\mathrm{PAL}}[\![e_2]\!] \xrightarrow{\mathrm{PAL}} l_1, \sigma_1; \mathbf{1}}\,(\mathrm{LAM}) \qquad D_1 \qquad \rho_0[x \mapsto l_6], \sigma_6 \vdash T_{\mathrm{PAL}}[\![e_2]\!] \xrightarrow{\mathrm{PAL}} v, \sigma_{new}; g_2}{\rho_0, \sigma_0 \vdash (\lambda x.T_{\mathrm{PAL}}[\![e_2]\!])\ (x_Y\ (\lambda x.\lambda y.T_{\mathrm{PAL}}[\![e_1]\!])) \xrightarrow{\mathrm{PAL}} v, \sigma_{new}; (\mathbf{1} \otimes g'') \oplus g_2}\,(\mathrm{APP})$$

$$\begin{array}{lllllllll}
\text{where } \rho_0 & = & \rho_I \cup \rho & \sigma_0 & = & \sigma_I \cup \sigma & g' & = & (\mathbf{1} \otimes \mathbf{1}) \oplus \mathbf{1} \\
\rho_1 & = & [x' \mapsto l_2] & \sigma_1 & = & \sigma_0[l_1 \mapsto \mathbf{cl}(\rho_0, x, T_{\mathrm{PAL}}[\![e_2]\!])] & g'' & = & (\mathbf{1} \otimes \mathbf{1}) \oplus g' \\
\rho_2 & = & \rho_1[y' \mapsto l_4] & \sigma_2 & = & \sigma_1[l_2 \mapsto \mathbf{cl}(\rho_0, x, \lambda x.T_{\mathrm{PAL}}[\![e_1]\!])] & g''' & = & (\mathbf{1} \otimes \mathbf{1}) \oplus g'' \\
\rho_3 & = & \rho_0[x \mapsto l_5] & \sigma_3 & = & \sigma_2[l_3 \mapsto \mathbf{cl}(\rho_1, y', e')] & & & \\
 & & & \sigma_4 & = & \sigma_3[l_4 \mapsto \mathbf{cl}(\rho_1, y', e')] & & & \\
 & & & \sigma_5 & = & \sigma_4[l_5 \mapsto \mathbf{cl}(\rho_2, z', y'\ y'\ z')] & & & \\
 & & & \sigma_6 & = & \sigma_5[l_6 \mapsto \mathbf{cl}(\rho_3, y, T_{\mathrm{PAL}}[\![e_1]\!])] & e' & = & x'\ (\lambda z'.y'\ y'\ z')
\end{array}$$

Figure 5.23: PAL derivation with Y-combinator. For readability, space costs are omitted, and the derivation tree is broken into five subtrees.

Usually, such constants are translated to abstractions. But this introduces a problem in an environment-based semantics: the translation on values picks an arbitrary environment, here the empty environment. If the PAL' constants are replaced by abstractions, evaluating the translated expression results in these abstractions "capturing" the local environment during evaluation, which in general cannot agree with the arbitrary choice for the values. Rather than providing a sufficiently general equivalence relation on closures, we avoid the problem by ensuring that our arbitrary choice of environment is the correct one.

Using a relation for the translation (as motivated for recursion) is effectively like having an equivalence relation on values, and we could take advantage of that here. But the given solution is simpler for this problem.

- It translates environments, values, and store-values relative to the appropriate store.

Aside from the recursion issue, using a relation also simplifies the translation on locations by allowing it to be independent of location names.

Theorem 5.1 now shows that the PAL model can simulate the PAL' model with only a constant factor of overhead. To prove this, Lemma 5.1 shows that the simulation holds for all contexts.

**Lemma 5.1 (Equivalence of PAL' and PAL)** *If $e$ evaluates in the PAL' model:*

$$\rho', \sigma', R' \vdash e \xrightarrow{\text{PAL}'} v', \sigma' \cup \sigma'_{new}; g', s'$$

*and for any context of $\rho$, $\sigma$, and $R$ for the corresponding PAL derivation such that*

- *its initial context is the translation of that of the PAL' derivation: $T_{\text{PAL}}[\![\rho']\!]_{\sigma'}[\![\rho]\!]_{\sigma}$, $T_{\text{PAL}}[\![\sigma']\!][\![\sigma]\!]$, $T_{\text{PAL}}[\![R']\!]_{\sigma'}[\![R]\!]_{\sigma}$, $S(R,\sigma) \leq k \cdot S(R',\sigma')$, and*
- *it uses the initial environment and initial store defined in Figure 5.21: $\rho_I \cup \rho$ and $\sigma_I \cup \sigma$,*

*then $e$'s translation evaluates in the PAL model:*

$$\rho_I \cup \rho, \sigma_I \cup \sigma, R \vdash T_{\text{PAL}}[\![e]\!] \xrightarrow{\text{PAL}} v, \sigma \cup \sigma_{new}; g, s$$

*such that*

- *it results in the translated value: $T_{\text{PAL}}[\![v']\!]_{\sigma' \cup \sigma'_{new}}[\![v]\!]_{\sigma \cup \sigma_{new}}$, and*
- *its costs are at most a constant factor more than those of the PAL' evaluation: $W(g) \leq k \cdot W(g')$, $D(g) \leq k \cdot D(g')$, and $s \leq k \cdot s'$, for some constant $k$.*

*Proof:* We prove this by induction on the structure of the PAL' evaluation derivation. We assume that the PAL' derivation holds and prove the PAL derivation and side conditions hold, using a case analysis on the last rule used in the PAL' derivation. The definition of the translation $\mathcal{T}_{\text{PAL}}[\![]\!]$ on environments and stores make most cases entirely straightforward.

The second condition on the PAL context holds inductively since, by definition, the domains of the initial environment and store are distinct from any other variables or locations. In most cases, the first conclusion holds by simple observation of the definition of the translation. The second conclusion holds by showing the translation introduces only a constant factor larger computation graph and a constant factor of extra closures.

**case CONST, $e = c$:** If the constant is an integer or $\mathbf{div}_i$, then $\mathcal{T}_{\text{PAL}}[\![c]\!] = c$, and the conclusion holds since the PAL' constant rule corresponds to the PAL constant rule.

Otherwise, $\mathcal{T}_{\text{PAL}}[\![c]\!] = l_c$, and the conclusion follows from the definition of the bindings of the initial environment and store. It holds since the PAL' constant rule corresponds to a PAL variable lookup, so $g' = g$ and $s < k \cdot s'$ for some constant $k$ determined by the size of the closures in $\sigma_I$.

**case LAM, $e = \lambda x.e'$:** The conclusion holds since the PAL' abstraction rule corresponds to the PAL abstraction rule.

**case VAR, $e = x$:** This follows from the definition of the translation of an environment, since by definition $\mathcal{T}_{\text{PAL}}[\![\rho'(x)]\!]_{\sigma'}[\![\rho(x)]\!]_{\sigma}$ holds if $\mathcal{T}_{\text{PAL}}[\![\rho']\!]_{\sigma'}[\![\rho]\!]_{\sigma}$ holds. The conclusion follows since the PAL' variable lookup corresponds exactly to a PAL variable lookup, of a value that is at most a constant factor larger than the PAL' value.

**case APP, $e = e_1\ e_2$:** By induction, the lemma holds for the two sub-derivations for $e_1$ and $e_2$. By assumption, the function $e_1$ evaluates to a closure, so there are three subcases, depending on whether this closure is recursive or not and which translation we use for a recursive closure.

If it is not recursive, we use the simplest closure translation, which does not introduce a binding for the dummy variable _. Recall that we omit bindings for this dummy variable in the translation of the environment. So, we can use induction on the body of the closure since the body of the PAL closure is the translation of the body of the PAL' closure. Thus, the conclusion holds since the PAL' application corresponds to a PAL application.

If it is recursive, in both cases we obtain a PAL closure that contains the translation of the body of the PAL' closure. We need this closure in the context during the evaluation of the function body to use induction. With the simpler closure translation, we immediately have such a closure, and the conclusion holds. With the complex closure translation, there are first four VAR, two LAM, and three APP steps before we obtain such a closure, as in Figure 5.24. But this still introduces only constant work, depth, and space overhead.

**case APPC,** $e = e_1\ e_2$**:** By assumption, the function $e_1$ evaluates to a constant $c$. By induction, the conclusion holds for both subexpressions, and in particular, we obtain the corresponding graphs $g_1$ and $g_2$. The exact structure of the PAL evaluation depends upon the constant $c$. Here we examine one of the simpler subcases, where $c = \mathbf{fst}$. Thus, by assumption, the argument evaluates to a pair. The PAL application of $T_{\text{PAL}}[\![c]\!]$ involves a constant amount of overhead—for **fst**, the overhead introduced by five VAR, two LAM, and three APP rules (*cf.* the binding for $l_{\mathbf{fst}}$ and the translation of pair store-values)—resulting in the graph in Figure 5.24. Similar results hold for each of the other constants. The exact structure of subgraph $g_3$ depends on the constant, but it is always of constant size.

**case PAIR,** $e = (e_1, e_2)$**:** By induction, the conclusion holds for the sub-derivations for $e_1$ and $e_2$. In the PAL model, $T_{\text{PAL}}[\![e]\!]$ evaluates to the translation of the pair $\langle v_1, v_2\rangle$. This evaluation takes one VAR, two LAM, and two APP steps (*cf.* the translation of pair expressions and the binding for $l_P$) in addition to the computation for the translated subexpressions, resulting in the graph in Figure 5.25. Thus it has only a constant factor more nodes and levels than the corresponding PAL' graph created by PAIR.

**case LETREC,** $e = \mathbf{letrec}\ x\ y = e_1\ \mathbf{in}\ e_2$**:** By induction, the conclusion holds for the subderivation. In the PAL model, $T_{\text{PAL}}[\![e]\!]$ evaluates to the translation of the appropriate recursive closure. Use of the Y-combinator represented by $x_Y$ encodes the recursion.

This evaluation takes two VAR, six LAM, and four APP steps (*cf.* the translation of **letrec** as detailed in Figure 5.23) in addition to the computation for $T_{\text{PAL}}[\![e_2]\!]$, resulting in the graph in Figure 5.25. Thus it introduces only a constant factor extra work, depth, and space. Note that neither $e_1$ nor its translation is evaluated until the function is used.

**cases IF-TRUE and IF-FALSE,** $e = \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3$**:** We show only the IF-TRUE case, as the IF-FALSE case is trivially different. For this case, $e_1$ evaluates to **true**.

By induction, the conclusion holds for both $e_1$ and $e_2$, and in particular, we obtain the corresponding graphs $g_1$ and $g_2$. In the PAL model, $T_{\text{PAL}}[\![e]\!]$ evaluates to the result of $T_{\text{PAL}}[\![e_2]\!]$. This evaluation takes one CONST, one VAR, three LAM, and three APP steps (*cf.* the translations for **if** expressions and the binding for $l_{\mathbf{true}}$) in addition to the computation for $T_{\text{PAL}}[\![e_1]\!]$ and $T_{\text{PAL}}[\![e_2]\!]$, resulting in the graph in Figure 5.25. Thus it introduces only a constant factor more work, depth, and space.

□

**Theorem 5.1 (Equivalence of PAL' and PAL)** *If $e'$ evaluates in the PAL' model:*

$$\cdot, \cdot, \{\} \vdash e' \xrightarrow{\text{PAL}'} v', \sigma'; g', s',$$

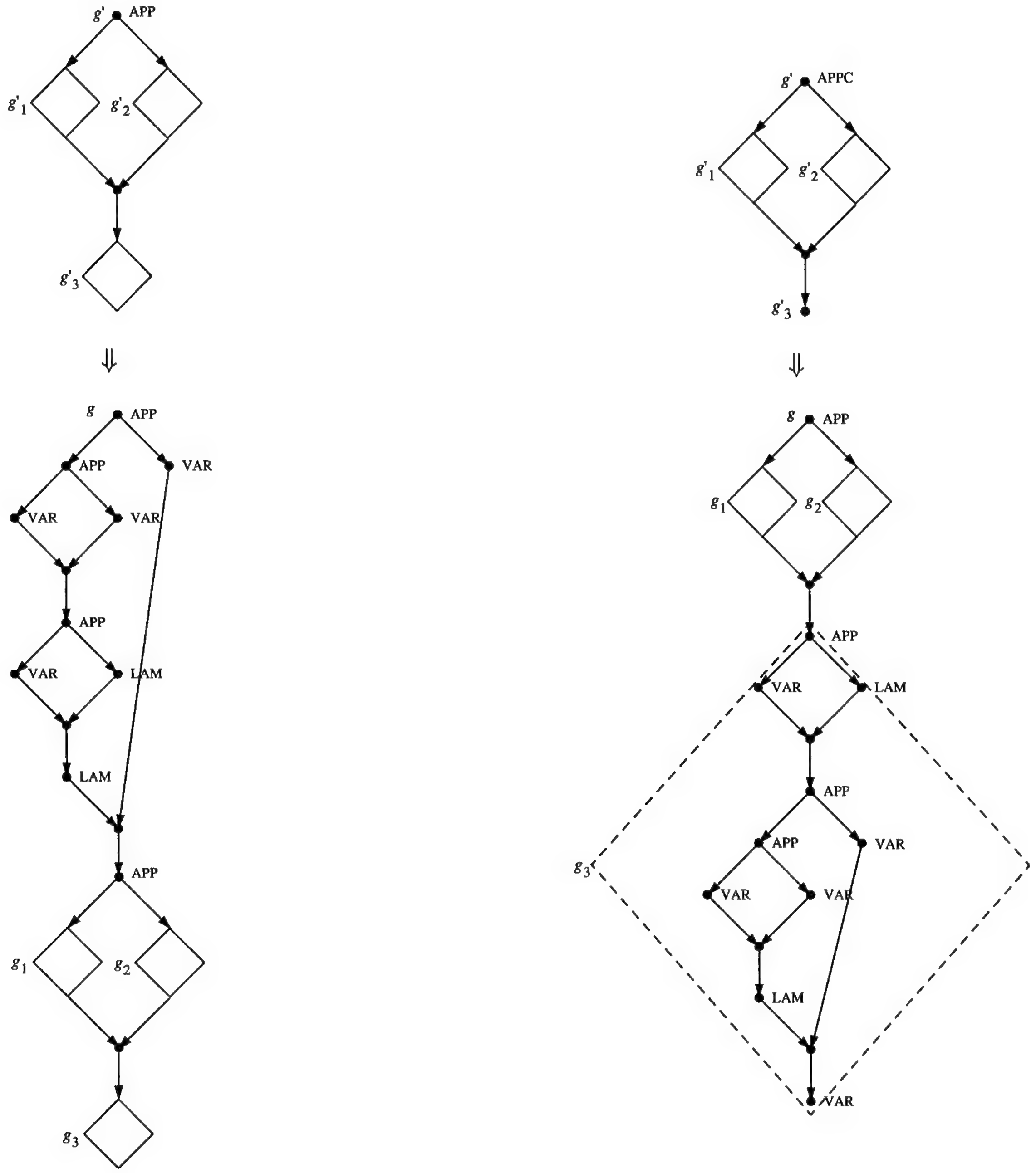


Figure 5.24: *Left:* The PAL computation graph $g$ corresponding to that produced by the PAL' APP rule, assumes the second translation of a recursive closure. *Right:* The PAL computation graph $g$ corresponding to that produced by the PAL' APPC rule. This example assumes $c = \mathbf{fst}$.

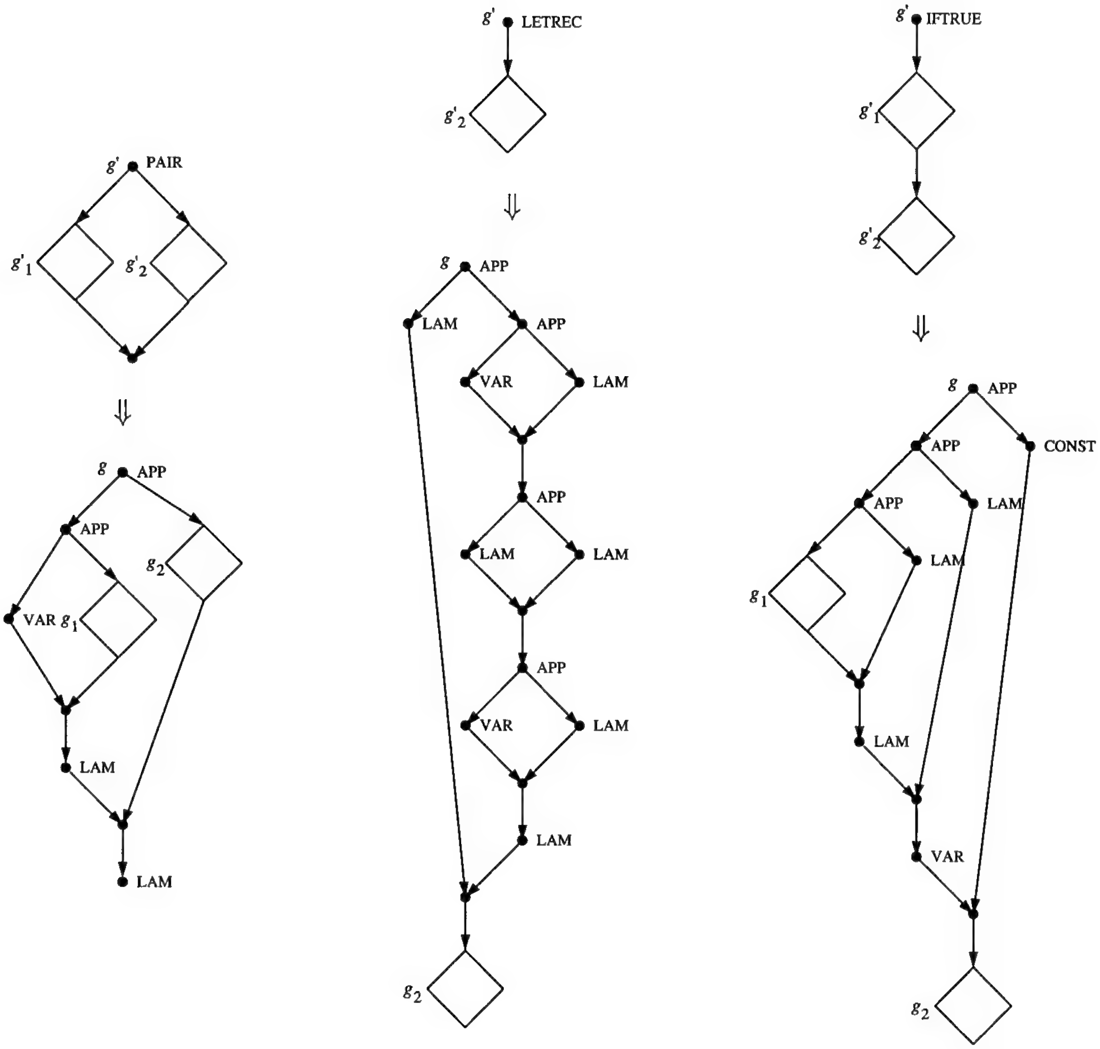


Figure 5.25: *Left:* The PAL computation graph $g$ corresponding to that produced by the PAL' PAIR rule. *Middle:* The PAL computation graph $g$ corresponding to that produced by the PAL' LETREC rule (as detailed by Figures 5.23 and 5.22). *Right:* The PAL computation graph $g$ corresponding to that produced by the PAL' IF-TRUE rule.

*and the corresponding* PAL *derivation uses the appropriately translated expression:*

$$e = (\lambda x_{\mathbf{add}} \ldots \lambda x_Y . T_{\mathrm{PAL}}[\![e']\!])\ (\lambda x.e_{\mathbf{add}})\ \ldots (\lambda x.e_Y)$$

*using the subexpressions defined in Figure 5.21, then the translation of $e'$ evaluates in the* PAL *model:*

$$\cdot, \cdot, \{\} \vdash e \xrightarrow{\mathrm{PAL}} v, \sigma; g, s$$

*such that*

- *it results in the translated value: $T_{\mathrm{PAL}}[\![v']\!]_{\sigma'}[\![v]\!]_{\sigma}$, and*
- *its costs are at most a constant factor more than those of the* PAL*' model: $W(g) \leq k \cdot W(g')$, $D(g) \leq k \cdot D(g')$, and $s \leq k \cdot s'$, for some constant $k$.*

*Proof:* This follows from Lemma 5.1. The initial applications in $e$ set up the initial environment and store for that lemma. □

# Chapter 6

# Intermediate model

The previous chapter shows how to define a simple language cost model, the PAL model, by defining its syntax and profiling semantics. Alone, however, a language model is limited in usefulness because it defines an abstract notion of evaluation costs. What we need is a relation between these abstract costs to the costs incurred in an implementation on a machine.

We now begin defining an implementation for the PAL model. As in a compiler, staging the implementation via an intermediate language or model frequently simplifies the problem. Here we stage the implementation using a parallel abstract machine model called the P-CEK$^q_{\mathrm{PAL}}$, loosely based on the serial CESK abstract machine [33].[1] It is relatively abstract since it uses many of the same semantic domains as the high-level language model, but it is more machine-like since it is based on a state transition relation. It also shows, at an abstract level, details such as how computation is scheduled onto processors.

The implementation effectively traverses the computation graph of the profiling semantics. An obvious idea is to make a *level-order traversal*, so that each level of the graph is executed in parallel on each step of the machine. However, this idea is not space-efficient: the space to store data corresponding to the graph nodes of a single wide level may dominate the computation. Instead, the machine implements a parallel generalization of a depth-first traversal, scheduling $q$ nodes at a time, where we relate $q$ to the number of processors $p$ on the machine.

Section 6.1 defines parallel traversals of graphs and reviews some previous results for them. Section 6.2 defines and explains the P-CEK$^q_{\mathrm{PAL}}$ machine. Section 6.3 shows the equivalence of the PAL model and the P-CEK$^q_{\mathrm{PAL}}$ machine. Later, Chapter 7 relates the P-CEK$^q_{\mathrm{PAL}}$ to less abstract machine models to complete the implementation.

[1] The CESK is one of many variants of the original SECD machine for implementing the $\lambda$-calculus [70]. The names of these machines are formed by the names of the meta-variables representing the elements of each state: the CESK uses a control string, environment, store, and continuation; the SECD uses a stack, environment, control string, and dump (a form of continuation).

## 6.1 Parallel Graph Traversals

Since we represent the computation as a graph of computation units, a specific traversal of the graph represents a scheduling of these computation units. The following definitions and theorems about traversals are either standard graph terminology or are from Blumofe and Leiserson [18] or Blelloch, Gibbons, and Matias [8]. Note that the definition of a graph traversal is somewhat different than is standard in that it requires all nodes to be visited.

### Graph traversals and schedules

**Definition 6.1 (Serial graph traversal)** *A* serial traversal of a graph *is a total ordering of the nodes such that, for each edge of the graph, the edge's source occurs earlier in the ordering than the edge's target. I.e., the ordering respects the dependencies of the graph, such that we can traverse a node only after all of its parents have been traversed.*

**Definition 6.2 (Parallel graph traversal)** *A* parallel traversal of a graph $g$ *is a sequence of* $k \geq 1$ *steps, where each step* $i$*, for* $i = 0, \ldots, k-1$*, defines a set of nodes,* $V_i$ *(that are* visited*, or scheduled, at this step), such that the following two properties hold:*

1. *Each node appears exactly once in the schedule: the sets* $V_0, \ldots, V_{k-1}$ *partition the nodes of* $g$*.*
2. *A node is scheduled only after all its ancestors have been: if* $n' \in V_i$ *and* $n$ *is an ancestor of* $n'$*, then* $n \in V_k$ *for some* $k < i$*.*

**Definition 6.3 ($q$-traversal)** *A* $q$-traversal of a graph $g$*, for* $q \geq 1$*, is a parallel traversal such that each step schedules at most* $q$ *nodes.*

Note that a serial traversal is simply a 1-traversal.

Consider a traversal $T = V_0, \ldots, V_{k-1}$ of $g$. A node $n$ of $g$ is *scheduled* prior to a step $i$ in $T$ if it appears in the traversal prior to step $i$, *i.e.*, $n \in V_0 \cup \cdots \cup V_{i-1}$. An unscheduled node $n$ is *ready* at step $i$ in $T$ if all its ancestors (equivalently, all its parents) are scheduled prior to step $i$. The *greedy $q$-traversal*, $T_q$ of a graph $g$, *based on* a 1-traversal of $g$, $T_1$, is the traversal that on each step $i$, schedules the $q$ earliest nodes in $T_1$ that are ready (or all the ready nodes in $T_1$ if there are fewer than $q$). In other words, for all ready nodes $n$ and $n'$, if $n$ precedes $n'$ in $T_1$, then either both are scheduled, neither are scheduled, or only $n$ is scheduled. Any sequence $P = V_0, \ldots, V_i$, for $i < k$ is a *prefix* of $T$.

Let $T_q$ be the greedy $q$-traversal based on a 1-traversal $T_1$. For each prefix, $P_q$, of $T_q$, consider the longest prefix, $P_1$, of $T_1$ that includes only nodes in $P_q$. We say a node is *premature with respect to $P_q$* if it is in $P_q$ but not in $P_1$.

Computation graphs are *dynamically unfolding* in that

- initially, only the root node is revealed;

- when a node is scheduled, its outgoing edges are revealed; and
- when all of the incoming edges of a node are revealed, the node is revealed and available for scheduling.

We consider only online scheduling algorithms for these graphs, *i.e.*, each step's scheduling decision is based on only the revealed graph nodes and edges. In other words, computation is scheduled at run time, not compile time.

**Definition 6.4 (Level-order traversal)** *A* level-order traversal of a graph *$g$ is an $\infty$-DFT based on a 1-DFT of the graph.*

## Costs of scheduling

**Theorem 6.1 (Premature nodes of greedy traversal [8])** *For any graph $g$ and any 1-traversal $T$ of the graph, the maximum number of premature nodes in the greedy $q$-traversal based on $T$ is at most $(D(g)-1)(q-1)$.*

**Theorem 6.2 (Steps of greedy traversal [18])** *For any graph $g$, a greedy $q$-traversal of $g$ takes at most $W(g)/q + D(g)$ steps.*

Note that Brent's Theorem [20] is a special case of Theorem 6.2 for level-order traversals.

**Definition 6.5 (Depth-first traversal)** *A* depth-first traversal *(DFT or 1-DFT) is obtained by maintaining a stack of ready nodes: the stack contains the root nodes in any order initially, and at each step, the top node is popped from the stack and scheduled. Then any newly ready nodes are pushed on the stack.*

**Definition 6.6 (Depth-first $q$-traversal)** *A* depth-first $q$-traversal *($q$-DFT) is a greedy $q$-traversal based on a 1-DFT.*

**Theorem 6.3 ($q$-DFT [8])** *For any series-parallel graph $g$, the following algorithm makes the $q$-DFT of $g$:*

> *Let $StA$ be an array initially containing the root node of $g$. Repeat the following two steps until all nodes in $g$ have been scheduled:*
>
> 1. *Schedule the first* $\min(q, |StA|)$ *nodes from $StA$.*
> 2. *Replace each newly scheduled node by its ready children, in left-to-right order, in place in the array $StA$.*

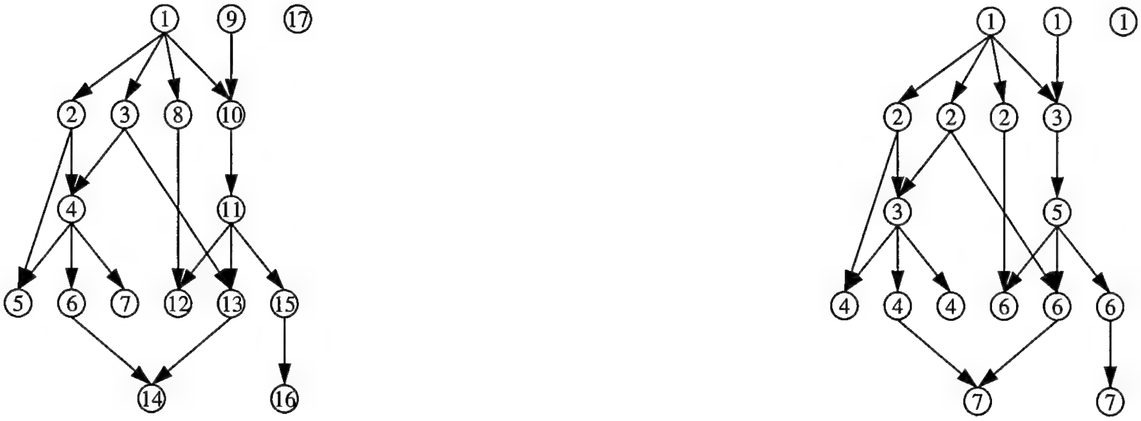


Figure 6.1: Example $q$-DFT. *Left:* Graph with 1-DFT ordering. *Right:* Same graph with 3-DFT ordering.

### Space costs

Assume that each node $n$ of a graph $g$ is associated with an integer $S_n(n)$ representing the net amount of space it allocates, or deallocates if the integer is negative. We assume that the amount of memory to allocate for each node is independent of the traversal. This includes memory for program variables and any temporary results of the computation. For deallocation, this assumption is too restrictive in languages with

- dynamic scheduling, when the deallocation of thread control information depends on which is the last to finish; or
- garbage collection, when the deallocation of a value depends on which thread references it last.

So we assume that the amount to deallocate is dependent on the traversal, *e.g.*, the memory for a value can be deallocated only after the last referencing thread finishes. Thus, the last node referencing the memory is credited for its deallocation.

**Definition 6.7 (Space of traversal prefix [8])** *For any prefix $P = V_0, \ldots, V_{i-1}$ of a $q$-traversal, the* space of the traversal, *written $S_P(P)$, is the size of the program input plus the space allocated by the nodes in the traversal, $\sum_{j=0}^{i-1} \sum_{n \in V_j} S_n(n)$.*

**Definition 6.8 (Space complexity of traversal [8])** *For a $q$-traversal $T = V_0, \ldots, V_{k-1}$, the* space complexity of the traversal, *written $S_T(T)$, is the maximum reachable space in use after any step of the traversal,* $\max_{j=0}^{k-1} S_P(V_0, \ldots, V_j)$.

**Theorem 6.4 (Space of $q$-traversal [8])** *If $s$ is the space complexity of the 1-traversal of graph $g$, then the space complexity of any $q$-DFT of the graph is bounded above by $s + O(D(g)q)$, including all bookkeeping space.*

## 6.2 P-CEK$^{q}_{\mathrm{PAL}}$ machine

This section defines the intermediate abstract machine for implementing the PAL model. We start with a general overview of the abstract machine, which is also applicable to variants to be introduced for other language models. We then give a formal definition of this specific abstract machine.

The machine performs a series of steps, each transforming a group of *active states* and a store into a new group of active states and new store for the next step. Figure 6.2 illustrates this process. Each of the currently active states represents a thread of computation which can be performed in parallel. The machine starts with a single active state representing the entire computation, and it ends when there is one active state left with the result value. Each state is used for computation on only one step—that step creates new states to perform any successive computation. *I.e.*, we think of creating new states and discarding old states rather than modifying states. Each step also uses a global store to not only keep track of the program's store contents, but also to record partial results of the computation.

Since the intermediate model is machine-like, different costs are of interest. Here we track three costs: the total number of states processed $Q$, the number of parallel steps $\psi$, and the maximum reachable space $s$. We will relate these costs to the computation graphs and maximum reachable space of the profiling semantics.

The currently active states represent those states whose computation can be performed now, *i.e.*, the ready nodes. But to maintain space-efficiency, on each step we *select* and use at most $q$ active states. In effect, there may be "too much" parallelism in a program, and we need to bound the number of active states, as the space to store them on a given step may dominate the space requirements of the program. In particular, we select the most recently created states so as to produce a $q$-DFT.[2] Thus, the active states are kept in a multi-stack (stack, for short), which allows pushes and pops of multiple states at once. Theorem 6.1 then allows us to bound the number of active states relative to the number that would be used on a serial implementation. We later relate $q$ to the number of processors available on the machine.

Each P-CEK$^{q}_{\mathrm{PAL}}$ step consists of one substep for computation and two substeps for communication and synchronization, as illustrated in Figure 6.3. The computation substep consists of a transition on each selected state, resulting in intermediate states, and the communication and synchronization substeps each consist of a transition on these intermediate states. The transitions of the computation substep resemble the corresponding rules of the profiling semantics. Each transition results in a bounded number (zero, one, or two) of new states to be active on the next step, as well as any updates to the global store. For example, an application state creates two states for evaluating the subexpressions. If a state's computation leads to no new states, that state's branch of the computation is finishing and needs to synchronize. Transitions on each of these states are independent and are to occur in parallel.

[2] An earlier presentation of this implementation was not concerned with space-efficiency, and so selected all active states on each step, producing a level-order traversal [9].

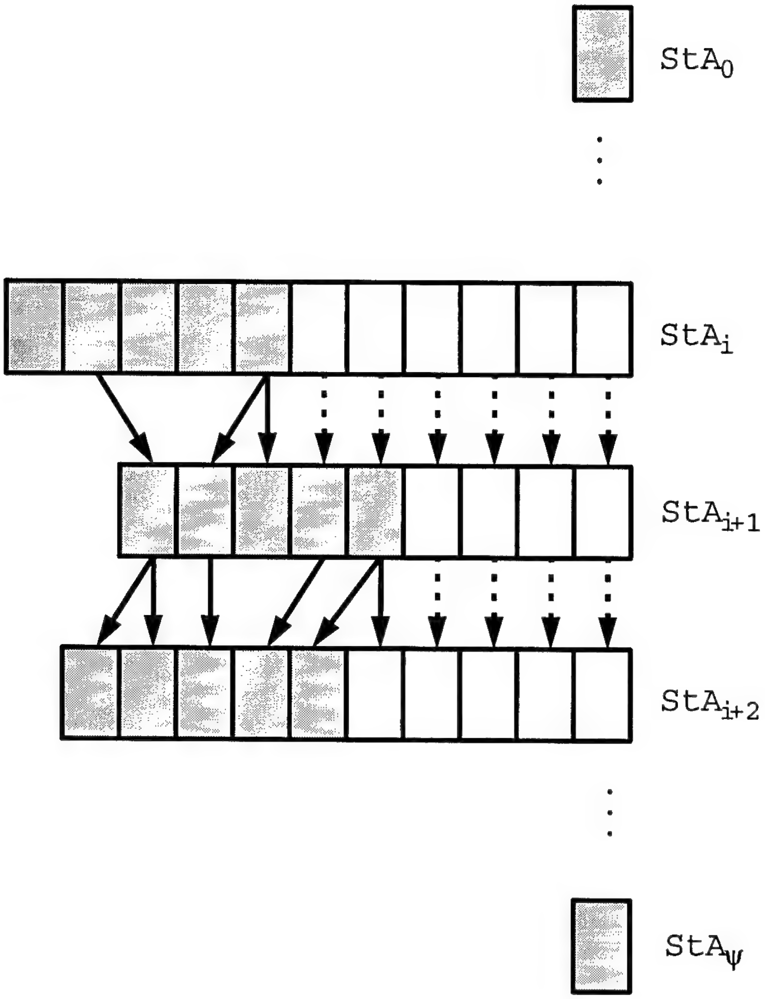
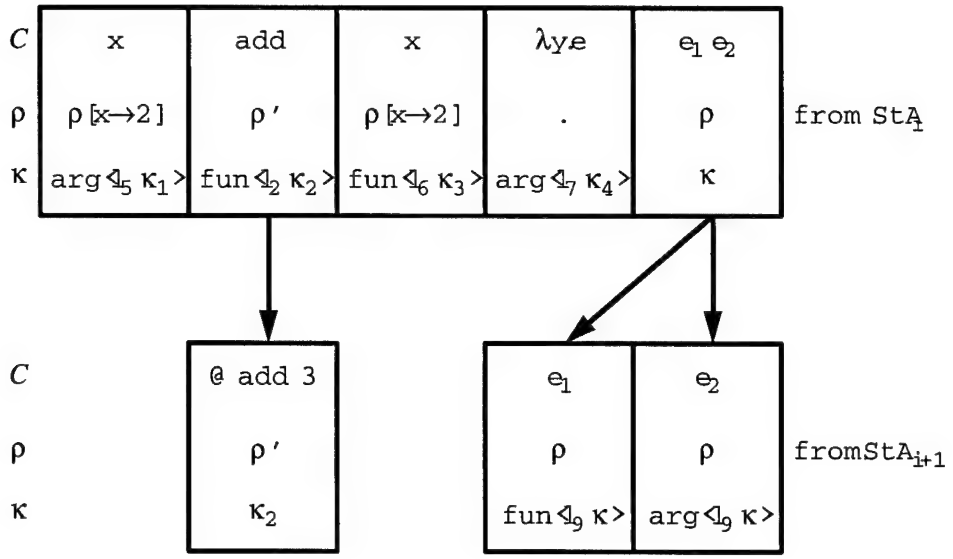


Figure 6.2: Illustration of P-CEK$^q_{\text{PAL}}$ active states during an evaluation. It starts with one active state representing the entire program and ends with one active state representing the result value. The states are kept in a stack. At most $q$ states are selected each step. Here, $q = 5$, and these selected states are shaded. These can create zero or more new states (solid arrows). Unselected states are still active in the next step (dashed arrows).



$$
\begin{array}{lrcl}
\text{where} & \sigma_i(l_2) & = & \mathsf{Val}\ 3 \\
& l_8, l_9 & \notin & \sigma_i \\
& \sigma_{i+1} & = & (\sigma_i \cup [l_8 \mapsto \mathbf{cl}(\cdot,y,e)][l_9 \mapsto \mathsf{Noval}]) \sqcup [l_5 \mapsto \mathsf{Val}\ 2][l_6 \mapsto \mathsf{Val}\ 2][l_7 \mapsto \mathsf{Val}\ l_8]
\end{array}
$$

Figure 6.3: Illustration of a P-CEK$^q_{\mathrm{PAL}}$ step. States with constants, variables, and abstractions finish evaluation immediately, but may synchronize with another computation and create a new state for the appropriate function body or constant function application. States with applications create two new states. The step may create and update synchronization locations.

The communication substeps synchronize the two parallel branches of an application when they finish. While such synchronization of states is clearly not independent, we parallelize each of these substeps.

Later chapters also define P-CEK variants for the implementation of the PSL and NESL models. Each has this basic structure of a series of steps, each selecting at most $q$ states to use in substeps for computation and then communications and synchronization. But these have significantly different substeps, especially for synchronization, and these differences affect most other parts of the machine, including the definition of a state. The following formal definition of the P-CEK$^q_{\mathrm{PAL}}$ is in a form that maximizes the similarity with these variants.

### Formal definition

A state *st* of the machine consists of a *control string* $C$, environment $\rho$, and *continuation* $\kappa$ (elsewhere, environments are sometimes denoted by $E$, and continuations by $K$, thus the

| | | | | | |
|---|---|---|---|---|---|
| $e$ | $\in$ | *Expressions* | ::= | $\ldots \mid @\ v_1\ v_2 \mid$ | application |
| | | | | **done** $v$ | final result |
| $C$ | $\in$ | *Controls* | ::= | $e$ | |
| $\rho$ | $\in$ | *Environments* | = | $Variables \xrightarrow{fin} Values$ | |
| $\kappa$ | $\in$ | *Continuations* | ::= | $\bullet \mid$ | program finishing |
| | | | | $\mathsf{fun}\langle l\ \kappa\rangle \mid$ | function finishing |
| | | | | $\mathsf{arg}\langle l\ \kappa\rangle$ | argument finishing |
| $st$ | $\in$ | *States* | ::= | $(C, \rho, \kappa)$ | |
| $St, StA$ | $\in$ | *StateArrays* | ::= | $\vec{st}$ | |
| $I$ | $\in$ | *IntermediateObject* | ::= | $St \mid$ | new states |
| | | | | $\mathsf{Fin}\langle v\ \kappa\rangle$ | finishing state |
| | | *ValueOpts* | ::= | $\mathsf{Noval} \mid \mathsf{Val}\ v$ | |
| $\sigma$ | $\in$ | *Stores* | = | $Locations \xrightarrow{fin} (StoreValues + ValueOpts)$ | |

Figure 6.4: P-CEK$^q_{\mathrm{PAL}}$ domains. The ellipses represent the expressions of Figure 4.1.

name "CEK"). The P-CEK$^q_{\mathrm{PAL}}$ does not include a store in the state, as does the CESK machine, but instead shares a single store for all states. This allows tracking the overall maximum reachable space, including any sharing among the states. The control string represents what is to be evaluated, the environment and store represent the context of the evaluation, and the continuation represents what to do after evaluation. In the P-CEK$^q_{\mathrm{PAL}}$, the control string is simply an expression, and the continuation records with which computations this one eventually synchronizes. Since the only synchronization is for pairs of states representing the evaluation of application function and arguments, the continuation keeps track of which application function or argument the state represents. Since applications may be nested, the continuation is effectively a stack. So, a state and its components are defined in Figure 6.4 where the ellipses represent the same expressions as in the PAL profiling semantics (*cf.* Figure 4.1).

We introduce two additional expressions beyond what is used in the profiling semantics. The expression $@\ v_1\ v_2$ represents the synchronization point after the function and argument evaluations and just before the function body evaluation. The expression **done** $v$ represents the result value of the computation.

As previously stated, each step of the machine starts with a group of active states and a store and produces a new group of active states and new store. Furthermore, these states are kept in a stack. Definition 6.9 defines this step relation. (See Chapter 3 for notation used with the stack of active states.)

**Definition 6.9 (P-CEK$^{q}_{\text{PAL}}$ step)** *A* step $i$ of the P-CEK$^{q}_{\text{PAL}}$ machine, *written*

$$StA_i, \sigma_i \overset{\text{PAL},q}{\hookrightarrow} StA_{i+1}, \sigma_{i+1}; Q_i, s_i,$$

*is defined in Figure 6.5. It starts with a stack of active states $StA_i$ and a store $\sigma_i$ and produces a new stack and store for the next step. This step processes $Q_i$ states and uses $s_i$ maximum reachable space.*

**Definition 6.10 (P-CEK$^{q}_{\text{PAL}}$ evaluation)** *In the P-CEK$^{q}_{\text{PAL}}$ machine, the* evaluation of expression $e$ to value $v$ starting in the environment $\rho$ and store $\sigma_0$, ends with store $\sigma_\psi$ and processes $Q$ states in $\psi$ parallel steps, using $s$ maximum reachable space, *or*

$$\rho, \sigma_0 \vdash e \overset{\text{PAL},q}{\Longrightarrow} v, \sigma_\psi; Q, \psi, s.$$

*For each of these $i \in \{0, \ldots, \psi - 1\}$ steps,*

$$StA_i, \sigma_i \overset{\text{PAL},q}{\hookrightarrow} StA_{i+1}, \sigma_{i+1}; Q_i, s_i,$$

*such that*

- *the machine starts with one active state for the whole program: $StA_0 = [(e, \rho, \bullet)]$, $\sigma_0 = \cdot$,*
- *the machine ends with one active state with the result value: $StA_\psi = [(\textbf{done}\ v, \cdot, \bullet)]$, and*
- *the total number of states processed and maximum reachable space are $Q = \sum_{i=0}^{m-1} Q_i$ and $s = \max_{i=0}^{m-1} s_i$.*

At the beginning of each step, the machine selects at most $q$ active states to evaluate on this step. Then the step uses three substeps in serial, one for computation using the $\overset{\text{PAL}}{\hookrightarrow}_{comp}$ transition, and two for communication and synchronization, using the $\overset{\text{PAL}}{\hookrightarrow}_{syncf}$ and $\overset{\text{PAL}}{\hookrightarrow}_{synca}$ transitions. For each selected state, the first transition results in either an array (of length at most two) of new states to be active on the next step, or it creates a special intermediate states $\mathsf{Fin}\langle v\ \kappa\rangle$ to indicate that a branch of an application has terminated with value $v$ and is ready for synchronization. The latter two substeps use the intermediate states for synchronization while passing through any regular states.

As shorthand, we say that the machine *processes* a state if the state is selected on some step of the machine. We assume that each state is unique, *e.g.*, by assuming that each expression has a unique label.

In the first substep, each of the selected states evaluates for one unit of computation:

$$\begin{array}{llllll}
\underline{\quad st \quad} & & & \underline{\quad I \quad} & & \underline{\quad \text{if/where} \quad} \\
(c, \quad -, \kappa) & - & \overset{\mathrm{PAL}}{\hookrightarrow}_{comp} & throw(c, \kappa) & \cdot & \\
(x, \quad \rho, \kappa) & - & \overset{\mathrm{PAL}}{\hookrightarrow}_{comp} & throw(\rho(x), \kappa) & \cdot & \\
(\lambda x.e, \; \rho, \kappa) & \sigma & \overset{\mathrm{PAL}}{\hookrightarrow}_{comp} & throw(l, \kappa) & [l \mapsto \mathbf{cl}(\rho', x, e)] & \rho' = restr(\rho, \lambda x.e), \; l \notin \sigma \\
(e_1 \; e_2, \; \rho, \kappa) & \sigma & \overset{\mathrm{PAL}}{\hookrightarrow}_{comp} & [(e_1, \rho, \mathsf{fun}\langle l \; \kappa\rangle), & [l \mapsto \mathsf{Noval}] & l \notin \sigma \\
 & & & \;(e_2, \rho, \mathsf{arg}\langle l \; \kappa\rangle)] & & \\
(@ \; l \; v, \; \cdot, \; \kappa) & \sigma & \overset{\mathrm{PAL}}{\hookrightarrow}_{comp} & [(e, \rho[x \mapsto v], \kappa)] & \cdot & \sigma(l) = \mathbf{cl}(\rho, x, e) \\
(@ \; c \; v, \; \cdot, \; \kappa) & \sigma & \overset{\mathrm{PAL}}{\hookrightarrow}_{comp} & throw(v', \kappa) & \sigma' & \delta(\sigma, c, v) = v', \sigma'; -
\end{array}$$

$$\begin{array}{llll}
\text{where} & throw(v, \bullet) & = & [(\mathrm{done}\; v, \cdot, \bullet)] \\
 & throw(v, \kappa) & = & \mathsf{Fin}\langle v \; \kappa\rangle \\
 & & & \\
 & restr(\; \rho, e) & = & \text{the environment } \rho \text{ restricted to the free variables in } e
\end{array}$$

$$\begin{array}{llllll}
\underline{\quad I \quad} & & & \underline{\quad I' \quad} & & \underline{\quad \text{if/where} \quad} \\
\mathsf{Fin}\langle v_1 \; \mathsf{fun}\langle l \; \kappa\rangle\rangle & \sigma & \overset{\mathrm{PAL}}{\hookrightarrow}_{syncf} & [] & [l \mapsto \mathsf{Val}\; v_1] & \sigma(l) = \mathsf{Noval} \\
\mathsf{Fin}\langle v_1 \; \mathsf{fun}\langle l \; \kappa\rangle\rangle & \sigma & \overset{\mathrm{PAL}}{\hookrightarrow}_{syncf} & [(@ \; v_1 \; v_2, \cdot, \kappa)] & \cdot & \sigma(l) = \mathsf{Val}\; v_2 \\
I & \sigma & \overset{\mathrm{PAL}}{\hookrightarrow}_{syncf} & I & \cdot & I \neq \mathsf{Fin}\langle - \; \mathsf{fun}\langle - \; -\rangle\rangle \\
 & & & & & \\
\mathsf{Fin}\langle v_2 \; \mathsf{arg}\langle l \; \kappa\rangle\rangle & \sigma & \overset{\mathrm{PAL}}{\hookrightarrow}_{synca} & [] & [l \mapsto \mathsf{Val}\; v_2] & \sigma(l) = \mathsf{Noval} \\
\mathsf{Fin}\langle v_2 \; \mathsf{arg}\langle l \; \kappa\rangle\rangle & \sigma & \overset{\mathrm{PAL}}{\hookrightarrow}_{synca} & [(@ \; v_1 \; v_2, \cdot, \kappa)] & \cdot & \sigma(l) = \mathsf{Val}\; v_1 \\
I & \sigma & \overset{\mathrm{PAL}}{\hookrightarrow}_{synca} & I & \cdot & I \neq \mathsf{Fin}\langle - \; \mathsf{arg}\langle - \; -\rangle\rangle
\end{array}$$

$$StA, \sigma \overset{\mathrm{PAL},q}{\hookrightarrow} (\mathop{+\!\!\!+} \vec{St}) +\!\!+ [st_{q'}, \ldots, st_{k-1}], \sigma'''; q', S_r(StA, \sigma)$$

$$\begin{array}{llllllll}
\text{if} & StA & = & [st_0, \ldots, st_{k-1}] & & & & \\
 & q' & = & \min(q, k) & \text{select at most } q \text{ states per step} & & & \\
 & st_i, \sigma & \overset{\mathrm{PAL}}{\hookrightarrow}_{comp} & I_i, \;\; \sigma_i & \text{for each } i \in \{0, \ldots, q'-1\} & \sigma' & = & \sigma \cup (\bigcup \vec{\sigma_i}) \\
 & I_i, \;\; \sigma' & \overset{\mathrm{PAL}}{\hookrightarrow}_{syncf} & I'_i, \;\; \sigma'_i & \text{for each } i \in \{0, \ldots, q'-1\} & \sigma'' & = & \sigma' \sqcup (\bigcup \vec{\sigma'_i}) \\
 & I'_i, \;\; \sigma'' & \overset{\mathrm{PAL}}{\hookrightarrow}_{synca} & St_i, \; \sigma''_i & \text{for each } i \in \{0, \ldots, q'-1\} & \sigma''' & = & \sigma'' \sqcup (\bigcup \vec{\sigma''_i})
\end{array}$$

Figure 6.5: Definition of the P-CEK$^q_{\mathrm{PAL}}$ abstract machine step. Assume all new locations of the computation step are chosen or renamed to be distinct.

- The cases for constants, variables, and abstractions correspond to those in the profiling semantics. Evaluating an abstraction creates a closure with an environment restricted to those variables free in the function body. This ensures that no extraneous data is live, so that we can prove our space bounds.

- Evaluating an application $e_1\ e_2$ creates two states, one to evaluate the function $e_1$ and one to evaluate argument $e_2$, which later steps can evaluate in parallel. The continuation of each new state indicates which branch it is: the function ($\mathsf{fun}\langle l\ \kappa\rangle$) or argument ($\mathsf{arg}\langle l\ \kappa\rangle$).

  The transition also creates this new location $l$ placed in these continuations. The location is used for synchronizing the two branches and to store the value of whichever branch finishes first. The location initially contains Noval to indicate that neither branch has finished.

- Evaluating an expression $@\ v_1\ v_2$ initiates evaluation of the function body or performs a constant application, as appropriate.

There is no transition for the expression **done** $v$, since the abstract machine stops after a state with that expression is created. This substep uses an auxiliary function $\mathit{throw}(v, \kappa)$ to use the value $v$ with continuation $\kappa$—if the continuation is empty, the entire computation is finished, otherwise this state needs to synchronize.

The synchronization substeps coordinate the function and arguments branches of an evaluation. When the first branch finishes, the machine updates the synchronization location to contain the result value of that branch. When the second finishes, the machine creates a state to evaluate the function body. These substeps have transitions for each case of whether the function or argument finishes first or second. These transitions are grouped into two substeps, rather than one, to handle the case where both branches finish on the same step. To avoid both determining that they are each the first to finish, one branch, that of the function, has priority and checks first. (Given an atomic test and set operation, we could combine these substeps.) Since these substeps can update store location bindings, the stores are combined with $\sqcup$. Note that only the synchronization location bindings are ever updated, and those at most once (from Noval to Val $-$).

Note that we do not define the abstract machine in the same style as the profiling semantics. State transition functions, such as the P-CEK$^{q}_{\mathrm{PAL}}$ machine, are a small-step style semantics and do not lend themselves to a concise big-step style semantics as do the language models.

### Costs of a step

As in the profiling semantics, the space cost is the maximum reachable space during the entire computation. Definition 6.11 defines the reachable space of each step, measuring all the values reachable from some set of roots, as before. Section 6.3 relates these costs to those of the profiling semantics.

$$
\begin{aligned}
L(StA) &= \bigcup_{(e,\rho,\kappa)\in StA} (L_\rho(\rho) \cup L_\kappa(\kappa)) \\
L_\rho(\rho) &= rng(\rho) \\
L_\kappa(\kappa) &= \{\}
\end{aligned}
$$

Figure 6.6: Definitions for the *root values* $L(StA)$ of a step of the P-CEK$^q_{\mathrm{PAL}}$ machine. This is a set of values, where labels act as roots into the store.

To formally define the reachable space during evaluation, we consider its two components: the *control space*, for the control information such as the active states and their continuation stacks, and the *store space*, for the elements in the store. We include the space for the synchronization locations in the control space (*e.g.*, $L_\kappa(\mathsf{fun}\langle l\ \kappa\rangle)$ does not add $l$ to the labels) even though they are are kept in the store so that the locations in the profiling semantics correspond exactly to those in the serial P-CEK$^1_{\mathrm{PAL}}$ machine.

**Definition 6.11 (Reachable space of P-CEK$^q_{\mathrm{PAL}}$ step)** *The* reachable space of a step $i$ of the P-CEK$^q_{\mathrm{PAL}}$ machine, *written $S_r(StA_i, \sigma_i)$, is the sum of*

- *the* active states space $S_A(StA_i)$ *for the active states, including their environments and continuations: the sum of $(1 + |dom(\rho)| + |\kappa|)$ for those states $(e, \rho, \kappa)$ in $StA_i$, where $|\kappa|$ is the length of the continuation stack $\kappa$; and*

- *the* store space $S_\sigma(StA_i, \sigma_i)$ *for program variables and all temporary values: equals the space in the store reachable from the active states used as roots, $S(L(StA_i), \sigma_i)$, where $S(-,-)$ and $L(-)$ are defined in Figures 5.14 and 6.6, respectively.*

**Example 6.1** *As an example of the execution of the P-CEK$^q_{\mathrm{PAL}}$, Figure 6.7 shows the active states at the beginning of each step of evaluating the expression* **add** (**add** 1 2) (**add** 3 4). *For lower values of $q$, the evaluation may take more steps, but it processes the same total number of states. For comparison, Figure 6.8 shows the computation graph of the corresponding profiling semantics evaluation, using the appropriate states' expressions as node labels. Observe that each of these executions is a $q$-DFT of the graph, and thus for $q \geq 4$, it is also a level-order traversal.*

| | $q \geq 4$ | |
|---|---|---|
| **Step** $i$ | expressions in $StA_i$ | $q'$ |
| 1 | $\underline{\textbf{add}\ (\textbf{add}\ 1\ 2)\ (\textbf{add}\ 3\ 4)}$ | 1 |
| 2 | $\underline{\textbf{add}\ (\textbf{add}\ 1\ 2),\quad \textbf{add}\ 3\ 4}$ | 2 |
| 3 | $\underline{\textbf{add},\quad \textbf{add}\ 1\ 2,\quad \textbf{add}\ 3,\quad 4}$ | 4 |
| 4 | $\underline{\textbf{add}\ 1,\quad 2,\quad \textbf{add},\quad 3}$ | 4 |
| 5 | $\underline{\textbf{add},\quad 1,\quad @\ \textbf{add}\ 3}$ | 3 |
| 6 | $\underline{@\ \textbf{add}\ 1,\quad @\ \textbf{add}_3\ 4}$ | 2 |
| 7 | $\underline{@\ \textbf{add}_1\ 2}$ | 1 |
| 8 | $\underline{@\ \textbf{add}\ 3}$ | 1 |
| 9 | $\underline{@\ \textbf{add}_3\ 7}$ | 1 |
| | **done** 10 | |
| | **States processed:** | 19 |

| | $q = 2$ | |
|---|---|---|
| **Step** $i$ | expressions in $StA_i$ | $q'$ |
| 1 | $\underline{\textbf{add}\ (\textbf{add}\ 1\ 2)\ (\textbf{add}\ 3\ 4)}$ | 1 |
| 2 | $\underline{\textbf{add}\ (\textbf{add}\ 1\ 2),\quad \textbf{add}\ 3\ 4}$ | 2 |
| 3 | $\underline{\textbf{add},\quad \textbf{add}\ 1\ 2},\quad \textbf{add}\ 3,\quad 4$ | 2 |
| 4 | $\underline{\textbf{add}\ 1,\quad 2},\quad \textbf{add}\ 3,\quad 4$ | 2 |
| 5 | $\underline{\textbf{add},\quad 1},\quad \textbf{add}\ 3,\quad 4$ | 2 |
| 6 | $\underline{@\ \textbf{add}\ 1,\quad \textbf{add}\ 3},\quad 4$ | 2 |
| 7 | $\underline{@\ \textbf{add}_1\ 2,\quad \textbf{add}},\quad 3,\quad 4$ | 2 |
| 8 | $\underline{@\ \textbf{add}\ 3,\quad 3},\quad 4$ | 2 |
| 9 | $\underline{@\ \textbf{add}\ 3,\quad 4}$ | 2 |
| 10 | $\underline{@\ \textbf{add}_3\ 4}$ | 1 |
| 11 | $\underline{@\ \textbf{add}_3\ 7}$ | 1 |
| | **done** 10 | |
| | **States processed:** | 19 |

Figure 6.7: P-CEK$^{q}_{\text{PAL}}$ evaluations for Example 6.1. The underlined expressions correspond to the selected states of each step.

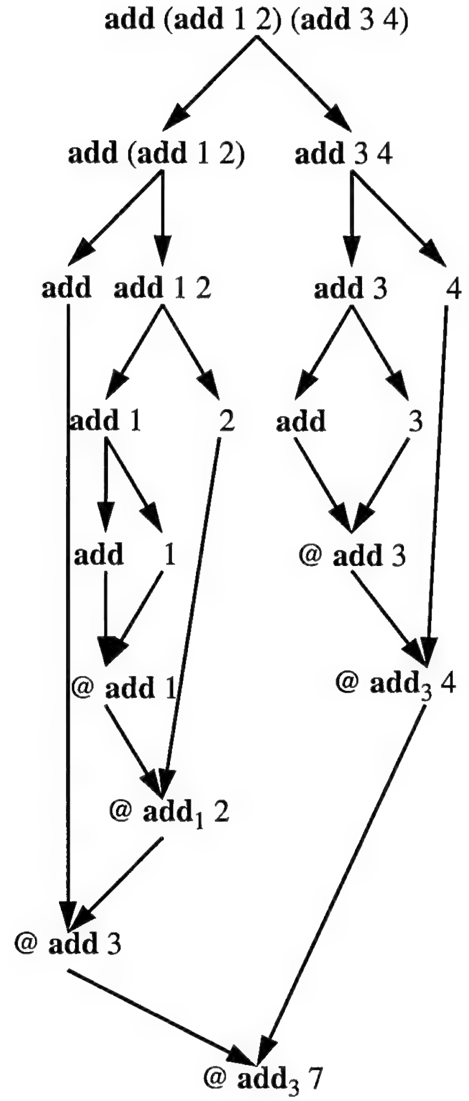


Figure 6.8: PAL computation graph for Example 6.1.

### Representation of environments

For any given program, the environments during its evaluation are of constant size (relative to the program size), bounded by the number of variables in the program. As a result, we treat the size of all environments as bounded by a constant. Since all values (constants and locations) are of constant size, any reasonable representation has constant time access and updates of environments.

In general, most environments are small, and treating them as of constant size is not unreasonable. In particular, the abstract machine restricts environments to a set of relevant free variables, and most functions use few variables.

In this implementation, any representation of environments must support the following operations:

- access of a variable's binding,
- extension with a new binding, possibly with a new variable, and
- restriction to a set of variables.

We could avoid the restriction operation by using a standard functional language compilation technique called *closure conversion* in the mapping to the intermediate model, but that would complicate the equivalence of Section 6.3. When examining the constant factors, or when not treating environments as having constant size, there is a efficiency tradeoff between different implementations for the costs of these operations. Some implementations use an array to make access constant time and restriction constant time per restricting variable, while making extension linear time (although lambda-lifting and compiler analysis can reduce the number of uses of extension). Alternatively, a list makes extension constant time extension, but access and restriction linear time. Balanced trees offer a middle ground.

Without the assumption that environments are of constant size, our time bounds need to be generalized to account for the time for environment accesses and updates. *E.g.*, representing environments as balanced binary trees, this adds a factor logarithmic in the number of distinct variables in the program [9]. We can then assume that the variables are renamed (*e.g.*, via deBruijn indices) so as to minimize the number of variables. Moreover, translating other language features into the basic $\lambda$-calculus syntax adds at most a constant number of variables (*e.g.*, Figure 5.20).

Without the assumption, our space bounds do not hold, since a computation node that updates an environment may allocate more than constant space. At worst, this multiplies the space bounds by the number of program variables, but a tighter bound may be possible by representing environments efficiently. For example, assuming environments are implemented as balanced binary trees that share bindings when possible, an update may allocate space logarithmic in the number of variables, duplicating some bindings so that the new environment is balanced. Naturally, each occurrence of any duplicated binding is eventually unreachable.

## 6.3 Equivalence of language and intermediate models

This section relates the P-CEK$^q_{\text{PAL}}$ to the PAL profiling semantics. In addition to proving its extensional correctness, we also prove bounds on the costs of the P-CEK$^q_{\text{PAL}}$ model as a function of those of the PAL model. In particular, we show the following:

**Serial:** The work, depth, and space required by the PAL model are within a constant factor of the number of states processed, steps, and space, respectively, of the P-CEK$^1_{\text{PAL}}$ (the machine that only selects one state per step).

**Parallel:** There is a one-to-one correspondence between states processed by the P-CEK$^q_{\text{PAL}}$ machine and nodes of the graph (*i.e.*, work) returned by the profiling semantics. Furthermore, we show that the P-CEK$^q_{\text{PAL}}$ machine executes a $q$-DFT of the graph. This allows us to use previous results on graph scheduling to show that the P-CEK$^q_{\text{PAL}}$ machine never schedules too many states prematurely relative to the P-CEK$^1_{\text{PAL}}$ machine. This, in turn, allows us to bound the extra reachable space required by the P-CEK$^q_{\text{PAL}}$ machine. It also allows us to bound the number of steps taken by the P-CEK$^q_{\text{PAL}}$ as a function of thePAL depth.

### Serial equivalence

To understand the space that is required to implement the P-CEK$^q_{\text{PAL}}$ machine we need to consider on each step both the space for any store values reachable via some label in the active states as well as space for the active states themselves. For each state we include the following space: for the control $C$, constant space; for the environment $\rho$, space proportional to the size of its domain; and for the stack of continuations $\kappa$, space proportional to the number of entries in the stack (here we are just accounting for space required by the active states stack itself and not for any values that are in the store). To find the root labels into the store we consider all labels accessible either though an environment or continuation of any of the substates.

**Theorem 6.5** (PAL **serial evaluation**) *If $e$ evaluates in the profiling semantics:*

$$\cdot, \cdot, \{\} \vdash e \overset{\text{PAL}}{\longrightarrow} v, \sigma; g, s,$$

*then it evaluates to the same result in the serial abstract machine:*

$$\cdot, \cdot \vdash e \overset{\text{PAL},1}{\Longrightarrow} v, \sigma'; Q, \psi, s',$$

*such that $s' \leq k \cdot s$, for some constant $k$.*

*Proof Outline:* To prove this we first generalize the statement to that of Lemma 6.1. There we consider the steps of P-CEK$^1_{\text{PAL}}$ required to evaluate an expression in some general context and bound the reachable space during those steps by the space specified by the profiling semantics plus the control space at the beginning of the evaluation. The theorem then holds by specializing the lemma to start with an empty environment, store, and roots, and one active state. □

**Lemma 6.1** *If $e$ evaluates in the profiling semantics:*

$$\rho, \sigma, R \vdash e \xrightarrow{\text{PAL}} v, \sigma'; g, s,$$

*and for any step $i$ of the serial abstract machine P-CEK$^1_{\text{PAL}}$,*

- *the machine starts with a state at the front of the active states stack that corresponds to this evaluation: $StA_i = [(e, \rho, \kappa)] +\!\!+ StA$, for some stack $StA$ and continuation $\kappa$;*
- *the semantics and machine can access the same locations: $L(StA_i) = R \cup \rho(FV(e))$; and*
- *these locations have the same values: $\forall_{l \in locs(L(StA_i), \sigma_i)} \sigma(l) = \sigma_i(l)$,*

*then*

- *on some future step $m \geq i$, the machine calls $throw(v, \kappa)$;*
- *the maximum reachable space is bounded by the space for the original active states, plus a constant factor more than the space of the profiling semantics: $\max_{j=i}^{m} S_r(StA_j, \sigma_j) \leq S_A(StA_i) + k \cdot s$ for some constant $k$.*

*Proof:* We prove this by structural induction on the language evaluation derivation and show a representative set of the cases. The remaining cases are similar.

**case VAR, $e = x$:** By the definition of the P-CEK$^q_{\text{PAL}}$ machine, *throw*$(v, \kappa)$ is called on step $i$, so $m = i$. And by VAR, $s = S(R \cup \{\rho(x)\}, \sigma)$, so

$$\begin{array}{lll} & \max_{j=i}^{m} S_r(StA_j, \sigma_j) & \\ = & S_A(StA_i) + S(L(StA_i), \sigma_i) & \text{(Definition 6.11)} \\ = & S_A(StA_i) + S(L(StA_i), \sigma) & \text{(3rd assumption)} \\ = & S_A(StA_i) + s & \text{(2nd assumption)} \end{array}$$

The other base cases, CONST and ABSTR, are similar.

**case APP, $e = e_1\ e_2$:** Alternately inspecting the machine rules and using induction, we obtain the following results about the executions of the subexpressions $e_1$ and $e_2$ and on the appropriate function body $e_3$. The steps of the P-CEK$^1_{\text{PAL}}$ corresponding to these three sub-evaluations are numbered $i_1$ to $m_1$, *etc.*, where $i_1 = i + 1$, $i_2 = m_1 + 1$, and

$i_3 = m_2 + 2$, and step $m_2 + 1$ is the appropriate function call transition. The active states at these important steps are

$$\begin{array}{rcl} StA_i & = & [(e_1\ e_2, \rho, \kappa)] +\!\!+ StA \\ StA_{i_1} & = & [(e_1, \rho, \mathsf{arg}\langle l\ \kappa\rangle)] +\!\!+ StA \\ StA_{i_2} & = & [(e_2, \rho, \mathsf{fun}\langle l\ \kappa\rangle)] +\!\!+ StA \\ StA_{m_2+1} & = & [(@\ l\ v_2, \cdot, \kappa)] +\!\!+ StA \\ StA_{i_3} & = & [(e_3, \rho[x \mapsto v_2], \kappa)] +\!\!+ StA \end{array}$$

Furthermore, these three sub-evaluations result in the appropriate values:

- $l$ is the value of $e_1$, where $\sigma_{m_1}(l) = \mathbf{cl}(\rho', x, e')$ and $\rho' = \mathit{restr}(\rho, e')$;
- $v_2$ is the value of $e_2$; and
- $v$ is the result of the function body, and thus of the entire application.

We now look at the reachable space during the evaluation. First look at the steps not in the inductive sub-evaluations, *i.e.*, steps $i_1$ and $m_2 + 1$. Examining the definition of the P-CEK$^q_{\mathrm{PAL}}$ machine and using Definition 6.11, we have

$$\begin{array}{rcl} S_r(StA_i, \sigma_i) & = & S_r(StA_{i_1}, \sigma_{i_1}) \\ S_r(StA_{m_2+1}, \sigma_{m_2+1}) & \leq & S_r(StA_i, \sigma_i). \end{array}$$

So the reachable space in these steps is not greater than in the others.

Now we look at the reachable space in the inductive sub-evaluations. Using induction we have

$$\max_{j \in \{i_{j'}, \ldots, m_{j'}\}, j' \in \{1,2,3\}} S_r(StA_j, \sigma_j) \leq \max_{j' \in \{1,2,3\}} (S_A(StA_{i_{j'}}) + k \cdot s_{j'}).$$

So we relate the control space at the beginning of these sub-evaluations, *i.e.*, $S_A(StA_{i_j})$, $j \in \{1, 2, 3\}$, to the control space of the starting step, $S_A(StA_i)$. For the first two sub-evaluations, $j \in \{2, 3\}$, we see that

$$S_A(StA_{i_1}) = S_A(StA_{i_2}) = S_A(StA_i) + 1.$$

For the space during the evaluation of the function body, $S_A(StA_{i_3})$, first observe that $|\rho'| + 2 \leq s_1$ by the definition of the store space since the closure with $\rho'$ must have been the result of a sub-derivation of $e_1$. Thus,

$$S_A(StA_{i_3}) + k \cdot s_3 \leq S_A(StA_i) + k \cdot s$$

and the conclusion holds.

The APPC case is similar, but somewhat simpler, since it does not involve induction for the function body.

□

### Parallel equivalence

Given the costs of serial execution in the abstract machine P-CEK$^1_{\text{PAL}}$, we are now concerned with the costs of parallel execution, for P-CEK$^q_{\text{PAL}}$ with any $q$. Parallel execution can require more space because it can create many more simultaneous parallel threads (*i.e.*, the active states stack can become much larger) and because it can have simultaneous access to many more locations in the store. We place bounds on how much extra space is needed.

As mentioned, the idea behind the proof is to show that the P-CEK$^q_{\text{PAL}}$ executes a $q$-DFT traversal of the computation graph returned by the semantics, then use the previous results on the number of nodes scheduled prematurely in a $q$-DFT [8] (*cf.* Section 6.1), and finally use these results to bound the space. By the machine traversing the graph we mean that there is a one-to-one correspondence between nodes in the graph and sets of a single step's computation, communication, and synchronization transitions for a given state. This implies that each parallel step of the P-CEK$^q_{\text{PAL}}$ selects $q'$ nodes of the graph, and the total number of states processed is equal to the size of the graph (*i.e.*, the work). The following lemma and theorem show that the machine evaluation corresponds to the specification of the computation graph.

We also state that the profiling semantics and abstract machine compute the same value. The proofs concentrate on intensional aspects—we could add details of the extensional equivalence, as in the proof of serial equivalence.

**Lemma 6.2 (P-CEK$^q_{\text{PAL}}$ executes traversal)** *If $e$ evaluates in the profiling semantics:*

$$\cdot, \cdot, \{\} \vdash e \overset{\text{PAL}}{\longrightarrow} v, \sigma; g, s,$$

*then it evaluates to the same result in the abstract machine:*

$$\cdot, \cdot \vdash e \overset{\text{PAL},q}{\Longrightarrow} v, \sigma'; Q, \psi, s'$$

*such that the machine executes a $q$-traversal of the profiling semantics' graph $g$. I.e.,*

- *the selected states and visited nodes correspond at each step, and*
- *the active states and ready nodes correspond at each step.*

*Proof Outline:* We prove this by induction on the steps of the machine. We could fully formalize this as in Lemma 6.1.

For brevity, we refer to states being visited or ready, rather than corresponding to nodes which are visited or ready, respectively. Clearly the initial state is ready, as it corresponds to the source of $g$.

Inductively, we need to show that any states added to the active states stack are ready on the next step—the non-selected states left in the stack remain ready. By a case analysis on the expression of each of the selected states, we see that the computation substep generates states corresponding to the graph.

Constants, variables, and abstractions finish immediately, thus this state corresponds to the unit graph specified for these expressions in the profiling semantics.

Applications generate two new states to start evaluating the subexpressions. These correspond to the two parallel children of the application node and are ready on the next step. Once both branches are scheduled and eventually finish, inductively, the machine generates a state for @ $v_1$ $v_2$ that is immediately ready, corresponding to the node before the function body. When selected, the machine starts evaluating the function body, inductively (for a user-defined function) or via $\delta$ (for a constant function). Thus the evaluation corresponds to the graph. □

**Corollary 6.1** *If $e$ evaluates in the profiling semantics:*

$$\cdot, \cdot, \{\} \vdash e \overset{\text{PAL}}{\longrightarrow} v, \sigma; g, s,$$

*then it also evaluates in the abstract machine:*

$$\cdot, \cdot \vdash e \overset{\text{PAL},q}{\Longrightarrow} v, \sigma'; Q, \psi, s'$$

*such that the number of states processed by the machine is the profiling semantics' work: $Q = W(g)$.*

*Proof:* This follows from the one-to-one correspondence of active states processed and nodes in the graph. □

**Theorem 6.6 (P-CEK$^q_{\text{PAL}}$ evaluates $q$-DFT)** *If $e$ evaluates in the profiling semantics:*

$$\cdot, \cdot, \{\} \vdash e \overset{\text{PAL}}{\longrightarrow} v, \sigma; g, s,$$

*then it also evaluates in the abstract machine:*

$$\cdot, \cdot \vdash e \overset{\text{PAL},q}{\Longrightarrow} v, \sigma'; Q, \psi, s'$$

*such that the machine executes a $q$-DFT of the profiling semantics' graph $g$.*

*Proof:* This follows since the machine selects $\min(q, |StA|)$ nodes per step and since $g$ is series-parallel, together with Theorem 6.3 and Lemma 6.2. □

**Corollary 6.2** *If $e$ evaluates in the profiling semantics:*

$$\cdot, \cdot \vdash e \overset{\text{PAL}}{\longrightarrow} v, \sigma; g, s,$$

*then it also evaluates in the abstract machine:*

$$\cdot, \cdot \vdash e \overset{\text{PAL},q}{\Longrightarrow} v, \sigma'; Q, \psi, s'$$

*such that the number of machine steps are bounded as a function of the profiling semantics' work and depth: $\psi \leq W(g)/q + D(g)$.*

*Proof:* This follows by Theorem 6.2. □

### Equivalence of space

Since the P-CEK$^q_{\text{PAL}}$ executes a traversal of the corresponding computation graph, we can use the machine to define the space costs of the graph nodes. Then using Theorem 6.4 we can bound the number of premature nodes on any given step of the P-CEK$^q_{\text{PAL}}$ and bound the memory used by these nodes, as Theorem 6.8 shows.

**Theorem 6.7** *Each step of a P-CEK$^q_{\text{PAL}}$ execution allocates at most $k$ space or deallocates at most $k$ space for each selected state, for some constant $k$.*

*Proof:* In the first substep, the cases calling *throw* create either one new state or one intermediate state (which is deallocated later in the same step and can be ignored). The abstraction case may also create a new restricted environment, which we assume to be of constant size, as discussed in Section 6.2. The other cases create at most two new states, one new environment binding, and two new continuations. Note that the $@\ l\ v$ case need not create an entirely new environment, as environments can be shared, as discussed in Section 6.2. The substep may also allocate at most one new store binding.

For each selected state, the second and third substeps deallocate any intermediate state created in the first substep. They may also allocate at most one new state or store binding. Note that which states create new states in this substep depends on the traversal.

Each selected state may also be credited with the deallocation of memory if this is the last state to reference it. This is a constant amount since each state refers to at most a constant amount of space. Note that we allow the crediting of a deallocation of a location even if it is still accessible, *i.e.*, in an environment. □

Since each step for a given selected state corresponds to a node (Lemma 6.2), each node allocates between $k$ and $-k$ space. By Theorem 6.5, the profiling semantics space is within a constant factor of the space complexity of the serial traversal. Thus as constant factors can be ignored, the profiling semantics space can be used in the context of Theorem 6.4 to provide a bound for the space of parallel execution.

**Theorem 6.8** (PAL **parallel space**) *If*

- *program $e$ evaluates in the profiling semantics: $\cdot, \cdot, \{\} \vdash e \xrightarrow{\text{PAL}} v, \sigma; g, s$; and*
- *thus the program computes in the abstract machine: $\cdot, \cdot \vdash e \overset{\text{PAL},q}{\Longrightarrow} v, \sigma'; Q, \psi, s'$,*

*then the maximum reachable space in the abstract machine is bounded by the maximum reachable space of the profiling semantics plus a function of the parallelism: $s' \leq k(s + D(g)q)$.*

*Proof:* Since the P-CEK$^q_{\text{PAL}}$ machine executes a $q$-DFT of $g$, then by Theorem 6.1, on any step of the P-CEK$^q_{\text{PAL}}$ there can be at most $D(g)q$ nodes executed prematurely relative to the P-CEK$^1_{\text{PAL}}$. Since each state transition in step $i$ of a P-CEK$^q_{\text{PAL}}$ machine adds at most constant space to the next state of the machine, then the proof is easy. In particular since the maximum reachable space taken by any step of the P-CEK$^1_{\mathrm{PAL}}$ is $k \cdot s$, and on any step of the P-CEK$^q_{\mathrm{PAL}}$ machine there are at most $D(g)q$ state transitions that were executed prematurely relative to some step of the P-CEK$^1_{\mathrm{PAL}}$ machine, each of which allocated at most constant space (Theorem 6.7), so the total space is $k(s + D(g)q)$. □

# Chapter 7

# Machine models

The previous chapter related the PAL profiling semantics to the P-CEK$^q_{\mathrm{PAL}}$ intermediate model. Now we need to complete the implementation of the language model by implementing the abstract machine on more standard machine models. Combining these two pieces results in the full implementation of the language.

Section 7.1 outlines the targeted parallel machine models. Section 7.2 discusses the implementation of the active state stack. These are then used in the overall implementation of the P-CEK$^q_{\mathrm{PAL}}$ abstract machine in Section 7.3. Throughout, the chapter uses standard sequence operations such as scans and reductions, which are summarized in Appendix A.

## 7.1 Machine models

Here we are most interested in implementing our parallel language models onto three specific traditional parallel machine models: the butterfly, hypercube, and Parallel Random-Access Machine (PRAM), as pictured in Figures 7.1, 7.2, and 7.3, respectively. Each consists of a set of processors connected by a communication network. The butterfly and hypercube are each based on specific network architectures used in practice, such that in each, any two processors are within $O(\log p)$ distance of each other. The PRAM is based on the unrealistic network architecture assumption that all processors are within constant distance of each other, *i.e.*, that communication is within a constant factor as fast as computation. But the PRAM is commonly used to describe algorithms so that computation issues are not obscured by communication issues—a common problem in more realistic models. These simpler PRAM algorithms can then be mapped to other models using standard implementation techniques [101, 123, 72]. We discuss several variants of the PRAM: primarily the concurrent-read concurrent-write (CRCW), but also the exclusive-read exclusive-write (EREW) and concurrent-read exclusive-write (CREW), which differ in what memory accesses are allowed, as their names imply.

We assume that in each model allocating an arbitrary-sized chunk of memory or accessing a memory location requires constant time. For the butterfly we assume that for $p$ processors

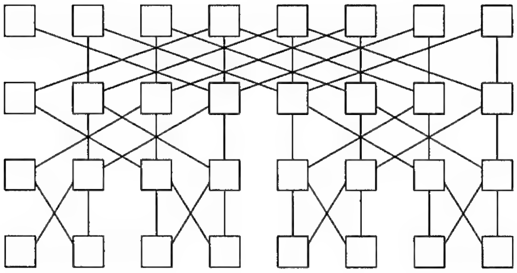


Figure 7.1: Illustration of butterfly network.

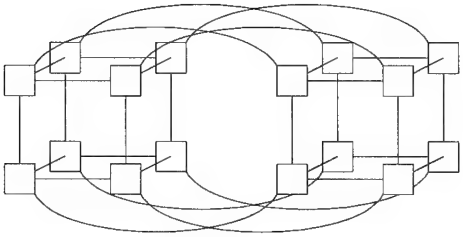


Figure 7.2: Illustration of hypercube network.

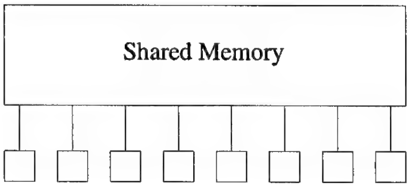


Figure 7.3: Illustration of Parallel Random Access Machine (PRAM).

| Machine | Randomized? | $TS(p)$<br>Time for scan |
|---|---|---|
| Butterfly | Yes | $O(\log p)$ |
| Hypercube | Yes | $O(\log p)$ |
| EREW PRAM | Yes | $O((\log p)^{3/2}/\sqrt{\log\log p})$ |
| CREW PRAM | Yes | $O(\log p \log\log p)$ |
| CRCW PRAM | No | $O(\log p/\log\log p)$ |

Figure 7.4: Time bounds $TS(p)$ for implementing scans and reductions on machines with $p$ processors.

we have $p\log_2 p$ switches and $p$ memory banks, and that memory references can be pipelined through the switches. On such a machine, each of the $p$ processors can access (read or write) $n$ elements in $O(n + \log p)$ time, with high probability [75, 101].[1] The $O(\log p)$ time is due to latency through the network. We also assume the butterfly network has simple integer adders in the switches, such that scan and reduce operations can execute in $O(\log p)$ time. A separate prefix tree, such as on the Connection Machine 5, would also be adequate. For the hypercube we assume a multiport hypercube in which messages can cross all wires on each time step, and for which there are separate queues for each wire. This model is quite similar to butterfly and has the same bounds for simulating shared memory. However, we do not need to assume that the switches have integer adders. We assume that primitive function calls can be implemented in the indicated amount of work (for the PAL, constant) on a single processor.

Our simulation uses the scan and reduce operations (*cf.* Appendix A), and our cost bounds are parameterized by their cost. We denote this time overhead $TS(p)$, as shown in Figure 7.4 [29, 100, 73, 41, 74]. Some of these bounds use randomized routing to avoid network congestion [122], and thus those bounds hold with high probability (w.h.p., for short).

We show that each step of a P-CEK$^{p\log p}_{\mathrm{PAL}}$ machine can be implemented in $O(TS(p))$ amortized time. These bounds hold with high probability (w.h.p.) on the randomized machines. The amortization comes from how we grow the active state stack. Since we have a bound on the number of steps required by the machine, this allows us to bound the total running time for these machines.

Note that in each model, $TS(p)$ dominates the latency for communication. Because of this, we do not have to separately parameterize our results by the latency.

[1] In this context, we mean that the time for network communication is within the specified bound with probability at least $1 - \frac{1}{n^k}$ for any constant $k$ and $n$ data to be transmitted across the network.

## 7.2 Representation of the active states multi-stack

The multi-stack (stack, for short) of active states requires three operations:

- creating a new stack at the beginning of an evaluation,
- pushing states onto the stack in parallel, and
- popping states from the stack in parallel.

We do not have a bound on the maximum size of the stack, so its representation must be able to grow.

We use an array-based representation of the stack for its constant-time lookup and update per element. To grow the stack, we create a new larger array when necessary, and copy the old elements into the new array. The key to efficiency is to copy infrequently, so that copying doesn't dominate the cost of using the stack. The standard technique for this is to double the size of the array each time it grows. The copying is sufficiently infrequent that its cost is amortized to constant time per step. Alternatively, for the PAL model, we can bound the number of states pushed or popped each step in terms of the number of states $q$ selected each step.

A stack of active states $StA$ is implemented by a *single-threaded dynamically growing array* (SDGA). A SDGA of states is a pair $(m, \vec{st})$ of its length $m$ and an array of states $\vec{St}$ such that

- the array is at least as large as the specified length: $m \leq |\vec{st}|$; and
- the rear of the array stores the SDGA's contents, so that it can grow at the front: data element $i$ of $StA$ is $st_{|\vec{st}|-m+i}$, for each $i \in \{0, \ldots, m-1\}$.

Each operation returns a new pair of the length and a new or modified array.

To initially create a stack of one state, create the pair $(m, \vec{st})$ where $m = 1$, $|\vec{st}| \geq 1$, and the state is in the last element of the array. A larger initial array would delay the need for creating a larger array as the stack grows. This clearly requires constant time and space.

We add elements to the stack at the end of each step. Each processor $i \in \{0, \ldots, p-1\}$ has states in an array $St_i$ to push onto the stack. In the PAL model, $|St_i| \leq 2$, but in the NESL model, the upper bound on the number of states is run-time dependent.

1. Compute (via an add-reduce operation) the number of states $k$ being pushed: $k = \sum_{i=0}^{p-1} |St_i|$.

   This requires $TS(p)$ time and $O(p)$ temporary space for the reduce operation.

2. If the array is not large enough for these new states ($k + m \geq |\vec{st}|$), create an array $\vec{st'}$ twice as big as the total number of states, *i.e.*, of size $2(k+m)$. This is large enough to hold all of the states and is also at least double the size of the original array. Then copy the contents of $\vec{st}$ into $\vec{st'}$ in parallel, such that each processor $i$ copies

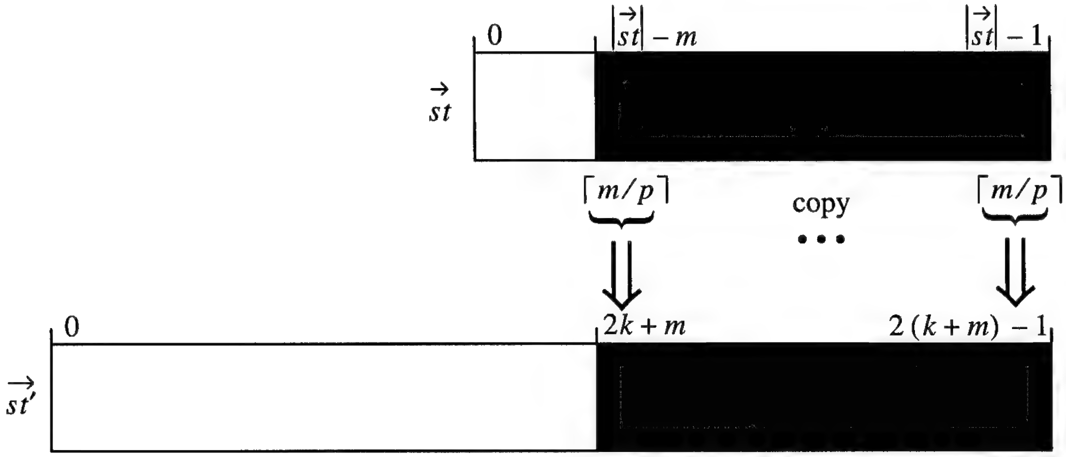


Figure 7.5: Step 2 of push operation on single-threaded dynamically growing array (SDGA).

a proportional share, *e.g.*, states $i\lceil m/p \rceil, \ldots, (i+1)\lceil m/p \rceil - 1$, (stored in locations $|\vec{st}| - m + i\lceil m/p \rceil, \ldots, |\vec{st}| - m + (i+1)\lceil m/p \rceil - 1$) as shown in Figure 7.5. From now on, ignore the old array and use the new array, *i.e.*, let the name $\vec{st}$ now refer to the array $\vec{st'}$.

The time for copying each of the $m$ elements is counted against the time for initially writing the elements that will be written into the array until the next time it grows. There are at least $m$ such elements, since the array doubles in size each time it grows. If the array doesn't grow again, the cost of this copy operation is counted instead against the initial writing of these elements. Thus, the time for copying data is at most twice that of initially writing data. This requires $O(k/p)$ amortized time and $O(k)$ space.

3. Move the new states into the array $\vec{st}$ such that the load is evenly distributed among the processors, as shown in Figure 7.6.

   (a) Each processor computes the starting point in array $St$ for its new states, and stores this in $\vec{i'}$. This can be accomplished by an add-scan, where each processor $i$ adds $|St_i|$ and gets the offset from the new top of the stack for its first state.

   This requires $O(TS(p))$ time and $O(p)$ temporary space for the scan operation and its result.

   (b) For each location in array $\vec{st}$ that receives a new state, *i.e.*, location $i \in \{|\vec{st}| - k - m, \ldots, |\vec{st}| - m - 1\}$, record the source of the state to be stored in the location. For example, the source for one location may be the $0_{\text{th}}$ element of $St_3$. Thus the sources are stored in an array $\vec{i''}$ of processor numbers (here, 3) and an array

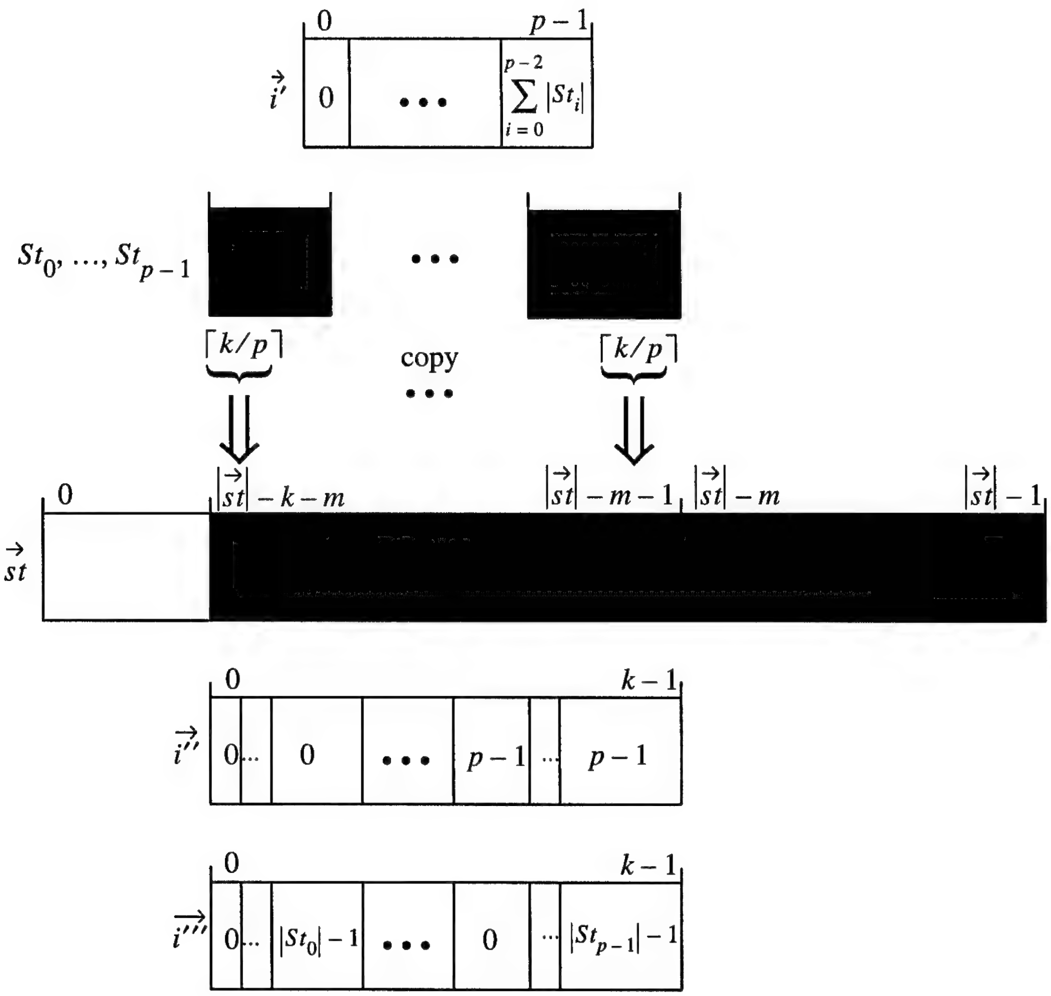


Figure 7.6: Step 3 of push operation on single-threaded dynamically growing array (SDGA). The cross-hatched section of $\vec{st}$ has its previous contents.

$\vec{i'''}$ of indices within the corresponding processor's states (here, 0). These can be computed by a segmented distribute of $\vec{i'}$ and a segmented index, respectively.

This requires $O(k/p + TS(p))$ time and $O(k)$ temporary space.

(c) Copy the states into array $\vec{st}$. Each processor $i$ copies a proportional share of the array, *e.g.*, states $i\lceil k/p\rceil, \ldots, (i+1)\lceil k/p\rceil - 1$ (to be stored in locations $|\vec{st}| - k - m + i\lceil k/p\rceil, \ldots, |\vec{st}| - k - m + (i+1)\lceil k/p\rceil - 1$), using $\vec{i''}$ and $\vec{i'''}$ to index into the appropriate arrays $St_j$.

This requires constant time per element, or $O(k/p)$ total time and no space.

So this step requires $O(k/p + TS(p))$ time and $O(k)$ temporary space.

Note that for the PAL model, the simpler alternative of each processor moving its array $St_i$ would evenly distribute the load since $|St_i|$ is bounded by a constant. But Chapter 9 uses SDGAs for models where that bound doesn't hold.

So in total, this requires $O(k/p + TS(p))$ amortized time and $O(k)$ space for the data, plus $O(p)$ temporary space. This temporary space can be reused in each step.

We remove elements from the stack when selecting (at most) $q$ states for each step. To pop $k$ states, each processor indexes into the array and grabs the appropriate $k/p$ states. We use a scan operation to assign tasks to processors and ensure that they are assigned to processors in order (*i.e.*, lower numbered processors get lower numbered states). Finally, the stack length is decremented by $k$. This requires $O(1)$ time and $O(k)$ space.

## 7.3 Implementation of steps

The stores of the abstract machine are all implemented with one global store that is mutated. The small stores resulting from the substeps of the abstract machine represent updates performed on the global store.

**Theorem 7.1 (Cost of P-CEK$^q_{\mathrm{PAL}}$ step)** *Each step of the P-CEK$^q_{\mathrm{PAL}}$ machine can be processed on a $p$ processor machine in $O(q/p + TS(p))$ amortized time (w.h.p., where appropriate) and $O(q)$ maximum reachable space on the butterfly, hypercube, and PRAM models.*

*Proof:* Each processor is responsible for at most $\lceil q/p\rceil$ of the current selected states, *i.e.*, processor $i$ is responsible for the states $[i\lceil q'/p\rceil, \ldots, (i+1)\lceil q'/p\rceil - 1]$, where $q' = \min(q, |StA|)$. We assume each processor knows its own processor number, so it can calculate a pointer to its section of the array.

The simulation of a step consists of the following phases, each of which we show can be executed with the given bounds:

1. Select the $q'$ states for this step, popping them from the stack.

   This requires $O(q'/p)$ time and $q'$ temporary space to copy the states.

2. Locally evaluate the states using the $\overset{\text{PAL}}{\hookrightarrow}_c$ transition. This requires accessing shared memory for reading but requires no communication among the states. Each transition requires constant time, returns an intermediate state of constant size, and allocates a constant amount of space. More precisely, the transition can be broken down so that all of the allocation is performed at once by using a add-scan operation to both determine how much space is needed and give each processor an index into a global array.

   Each processor makes a total of $O(q'/p)$ memory requests. The time for this on the CREW and CRCW PRAM is therefore $O(q'/p)$ (w.h.p., on the CREW). The time on the butterfly and hypercube is $O(q'/p + \log p)$ w.h.p. since the memory references require a $\log_2 p$ latency through the network. On any of these machines, this is bounded by $O(q'/p+TS(p))$ time. Since at most two new states and one new location are created per selected state, this requires $O(q')$ space: $O(q')$ control space and $O(q')$ temporary space for the intermediate states.

3. Locally evaluate the $\overset{\text{PAL}}{\hookrightarrow}_{s1}$ and $\overset{\text{PAL}}{\hookrightarrow}_{s2}$ substeps, synchronizing all processors between the two transitions. The "returned" stores are implemented as updates. Each processor can perform an update independently since each location appears in at most one function continuation and one argument continuation.

   Again, each transition accesses constant memory and allocates constant space. In fact, allocation can be eliminated by reusing the state that just ended this application's function or argument branch. If we avoid allocation, this phase requires $O(q'/p+TS(p))$ time (w.h.p., where appropriate) and constant temporary space.

4. Push the states created during this step onto the active state stack. This requires $O(q'/p + TS(p))$ amortized time (w.h.p., where appropriate) and $O(q')$ control space.

Adding the bounds for the three phases, we get the stated bounds for each of the machines. □

To account for memory latency in the butterfly and hypercube, and for the latency in the scan operation for all three machines, we process $p \cdot TS(p)$ states on each step instead of just $p$ (*i.e.*, we use a P-CEK$_{\text{PAL}}^{p \cdot TS(p)}$ machine).

**Corollary 7.1** *Each step of the P-CEK$_{\text{PAL}}^{p \cdot TS(p)}$ machine can be simulated within $O(TS(p))$ amortized time on the $p$ processor butterfly, hypercube, and PRAM machine models (w.h.p., where appropriate).*

**Corollary 7.2** *If $e$ evaluates in the profiling semantics:*

$$\cdot, \cdot, \{\} \vdash e \overset{\text{PAL}}{\longrightarrow} v, \sigma; g, s,$$

*then the abstract machine evaluation*

$$\cdot, \cdot \vdash e \overset{\text{PAL}, p \cdot TS(p)}{\Longrightarrow} v, \sigma'; Q, \psi, s'$$

*can be simulated within $O(W(g)/p + D(g)TS(p))$ amortized time and $O(s + p \cdot D(g)TS(p))$ maximum reachable space on the p processor butterfly, hypercube, and PRAM machine models (w.h.p., where appropriate).*

*Proof:* Theorem 6.6 relates the graph $g$ to the P-CEK$^q_{\mathrm{PAL}}$ computation, where $q = p \cdot TS(p)$. Theorem 6.2 bounds the number of steps of the graph traversal. There are $O(w/q + d)$ steps, and each step takes $O(TS(p))$ amortized time (w.h.p., where appropriate). Theorem 6.4 provides the space bound. □

## An improvement

These time results can be slightly improved on the CRCW PRAM. If we change the representation of the active state stack to allow a constant factor of holes in the array, we can replace the scan operations with *linear approximate compaction*: given an array of $m$ cells, $m'$ of which contain an object, place the $m'$ objects in distinct cells of an array of size $k \cdot m'$ for some constant $k > 1$ [73]. Gil, Matias, and Vishkin [42] have shown that the linear approximate compaction problem can be solved on a $p$ processor CRCW PRAM (ARBITRARY) in $O(m/p + \log^* p)$ expected time, using a randomized solution. Goldberg and Zwick [43] have shown that the problem can be solved deterministically in $O(m/p + \log\log p)$ time.

Since each selected state leads to at most two new states, the idea is to first allocate two new positions for each state, mark the states that will remain and then do an approximate compaction. This means that each processor is responsible for at most $k \cdot q/p$ states. The scan operations used for allocating space are also replaced by linear approximate compaction.

This breaks the lock-step simulation of the abstract machine. Each machine step selects at most $q$ elements of the array, but since the array has holes in it, this probably results in less than $q$ states (on average, $q/k$). Rather than requiring exactly $D(g)$ steps, the simulation takes at most $k \cdot D(g)$ steps.

Thus the time bounds for the overall simulation on the CRCW PRAM are $O(w/p + D(g)\log^* p)$ amortized w.h.p., if randomized, or $O(w/p + D(g)\log\log p)$ amortized, if not. The space bounds are unchanged, as a constant factor extra space is used throughout the simulation.

# Part III

# Other Models

# Chapter 8

# Speculative models

This chapter discusses a different model of parallelism, the Parallel Speculative $\lambda$-Calculus (PSL) model. It shares the same extensional semantics as that of the PAL model, but parallelizes computation differently. Like the PAL model, it evaluates the function and argument of an application in parallel. But it relaxes the synchronization constraint prior to evaluating the body, allowing more parallelism. If the function expression evaluates to a closure, evaluation of the function body starts immediately even if the argument expression has not finished evaluating. If the argument's value is needed and not yet computed, the lookup blocks until the value is computed.

For example, consider the evaluation of $(\lambda x.1)\ e$. The function finishes evaluating immediately, so the application proceeds while $e$ might be still evaluating. Thus the program quickly returns the value 1, even though the argument $e$ might still be evaluating. *In the* PSL *model, an expression might result in a value before the expression finishes evaluating.*

If the program evaluates without ever needing a subexpression's value, as is the case with $e$ in previous example, that subexpression is *irrelevant* [40, 48]. Here we describe two variants of the model: *full speculation*, which evaluates all expressions, even if irrelevant, and *partial speculation*, which can abort and discard irrelevant computations. The latter offers a wide spectrum of implementations, depending on which computations it aborts and when.

Speculative evaluation is most closely related to the futures of Multilisp and its many descendant languages [50, 51, 26, 31, 79, 69, 58, 125], the lenient evaluation of Id and pH [120, 87, 88], and parallel graph reduction, *e.g.*, [55, 96, 61].

- In Multilisp, any expression can be designated as a future that spawns a thread which may be executed in parallel. If its value is needed and not yet computed, the thread requesting the future's value blocks until the value is available. A future can be explicitly *touched* to force its evaluation and its synchronization with the touching thread. It can also be explicitly aborted—if its value is relevant, this leads to an error. Speculative evaluation is equivalent to designating all expressions as futures and disallowing touching. Full speculation also disallows aborting futures, whereas partial speculation allows aborting them, but in a safe manner not in the programmer's control.

- Full speculation and *leniency* are essentially the same thing, although the term "leniency" originally implied a specific lack of evaluation ordering [120]. Id and pH evaluate all subexpressions fully because they may contain side-effects, although a compiler might optimize cases when this is not necessary.

- Graph reduction is one technique for implementing lazy (call-by-need) functional languages. But since lazy evaluation entails an inherent lack of parallelism [64, 121], parallel versions of these languages have incorporated partial speculation, compromising on the laziness of the language.

The PSL model is "speculative" in two senses. First, it is speculatively parallel relative to the PAL model, as it allows a function body and argument to be evaluated in parallel when possible. This is consistent with Hudak and Anderson's *call-by-speculation* [53] terminology, which they contrasted with call-by-value and call-by-need. Second, it is speculative relative to a call-by-need evaluation, as it at least starts the evaluation of an argument even if it is irrelevant.

This contrasts with some descriptions of speculativeness [90, 37, 83] that are speculative relative to a call-by-value evaluation. By definition of those descriptions, the parallel execution of a program must be extensionally equivalent to the serial execution, even in the presence of control escapes or side-effects. *E.g.*, when evaluating **let** $x = e_1$ **in** $e_2$, if $e_1$ has an error or escapes, then any escapes or side-effects of $e_2$'s evaluation must be ignored. Thus, $e_1$ is considered *mandatory*, while $e_2$ is *speculative*. Similarly, any side-effects in $e_1$ must occur before any conflicting side-effects in $e_2$. Computation of $e_1$ and $e_2$ may still be parallelized within these constraints. Our results are still applicable to this alternate view of speculation by simply reversing which evaluations are considered mandatory or speculative.

The implementation of speculation is based on the idea of *suspending* threads. If one thread $\tau_1$ requests the value of another thread $\tau_2$ before the value is available, the machine suspends $\tau_1$. When $\tau_2$ finishes with a result value, $\tau_1$ reactivates and then accesses the value.

We implement suspension using a queue per thread of the threads suspended on it. *I.e.*, the above $\tau_1$ would be on $\tau_2$'s queue. Previous implementations, *e.g.*, [90, 87, 88], serialized the operations on these queues. Thus if other threads also suspend on $\tau_2$ at the same step of the machine that $\tau_1$ does, it would take steps proportional to the number of these threads to enqueue them. Similarly, it would take steps proportional to the number of all threads in the queue to reactivate them when $\tau_2$ finishes. However, the implementation here parallelizes these operations so that enqueuing and dequeuing multiple threads at once requires constant steps.

Whereas the PAL implementation is based on the scan and reduce operations, the PSL implementation uses a generalization, called fetch-and-add (*cf.* Appendix A). This operation allows efficient implementation of the queuing operations on multiple queues at once. Our cost bounds are parameterized by the time cost of this operation, $TF(p)$, as Figure 8.1 shows for the same machine models as in Section 7.1. These bounds hold with high probability (w.h.p., for short) [100, 73, 41, 74].

| | | $TF(p)$ |
|---|---|---|
| Machine | Randomized? | Time for fetch-and-add |
| Butterfly | Yes | $O(\log p)$ |
| Hypercube | Yes | $O(\log p)$ |
| EREW PRAM | Yes | $O((\log p)^{3/2}/\sqrt{\log\log p})$ |
| CREW PRAM | Yes | $O(\log p \log\log p)$ |
| CRCW PRAM | Yes | $O(\log p/\log\log p)$ |

Figure 8.1: Time bounds $TF(p)$ for implementing fetch-and-add on machines with $p$ processors.

We show that each step of a P-CEK$_{\mathrm{PSLf}}^{p\log p}$ machine can be implemented in $O(TF(p))$ amortized time, w.h.p. The amortization comes from how we grow the active state stack. Since we have a bound on the number of steps required by the machine, this allows us to bound the total running time for these machines.

Section 8.1 defines the PSL model, including its computation graphs and profiling semantics. Section 8.3 defines a generalization of the dynamically growing array to implement collections of threads. Sections 8.2 and 8.4 describe a fully speculative implementation and its costs. And finally Section 8.5 describes a partially speculative implementation and its costs. Neither of these implementations make a DFT of the computation graph, so we present no results on the space required by the implementations. However, it does make a greedy traversal, which we use in showing time bounds.

## 8.1 Language and Profiling semantics

Again for the sake of simplicity, we use the basic $\lambda$-calculus syntax of Chapter 4. We claim without proof that the translation of Section 5.5.2 preserves the equivalent of Theorem 5.1 for this model as well. *I.e.*, basing the PSL model on the extended $\lambda$-calculus results in the same asymptotic bounds. This holds for the work cost since it is equivalent to that of the PAL model, as Theorem 10.9 shows. Like for the PAL, it holds for the depth cost since the translation adds only a constant factor work overhead, and thus can only add a constant factor depth overhead. However, note that the depth overhead might be different than that for the PAL model.

This equivalence means that standard translations preserves speculativeness. For example, defining

$$\mathcal{T}_{\mathrm{PSL}}[\![\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2]\!] = (\lambda x.e_2)\ e_1$$

results in the two subexpressions $e_1$ and $e_2$ being evaluated in parallel. Data structures such as lists would be based on pairing—*e.g.*, a cons-cell would be a triple (nested pair) of a tag and the cell's contents. Using the encoding for pairs in Section 5.5.2 results in each pair being built speculatively in parallel. Thus any data structure using pairs is also built speculatively.

Since we include no basic serialization construct in the core of PSL, providing a serializing binding construct, for example, is more difficult. But it can be encoded using a *continuation passing style (CPS)* transformation (*e.g.*, [99]):

$$\mathcal{T}_{\mathrm{PSL}}[\![\mathbf{slet}\ x = e_1\ \mathbf{in}\ e_2]\!] = \mathit{CPS}[\![e_1]\!]\ (\lambda x.e_2).$$

Traditionally used for serial computation, CPS makes the standard serial path of evaluation control explicit. The transformation to CPS introduces additional dependences, so that no significant computation can be performed in parallel under speculative evaluation. Alternatively, we could simply add a special expression with a serial semantics.

Section 8.1.1 defines the computation graphs of the model. Then Section 8.1.2 defines its profiling semantics. Section 8.1.3 discusses using implicit recursion (as in the basic λ-calculus) versus explicit recursion (as in the extended λ-calculus) for the model.

### 8.1.1 Computation graphs

Relaxing the synchronization constraint during an application significantly affects the forms of computation graphs. To model that a computation may result in a value before its terminates, we distinguish a graph's *minimum* sink, when it results in a value, and its *maximum* sink, when it terminates (if it does). The depth of the minimum sink is denoted $D(g)$ and that of the maximum sink is $D'(g)$. Clearly by definition, the maximum sink is at a depth at least as great as that of the minimum sink, as the names imply. Since we now distinguish a source and two sinks, we draw a graph as a triangle, as illustrated in Figure 8.2.

It can be intuitively helpful to distinguish two classes of edges. The edges for applications are "control" edges, and the edges for variable lookups are "data" edges. These names are meant only for intuition since each edge represents a control dependence and allows for the flow of data. Each control edge corresponds exactly to an edge in the PAL computation graph for an application.

The form of an application's graph differs from that of the PAL model in several ways. There is still a single node to start the application, with edges to the subcomputations, but there is no single node for synchronization. There is also still an edge from the function's graph to that of the function body, but the existence of edges from the argument's graph depends on whether its value is accessed. In an application, edges may connect the interior nodes of the subgraphs, from either the function or argument subgraph to the function body subgraph. What edges exist depend on the expressions:

- Edges from the argument value (*i.e.*, the argument's minimum sink) connect to each of its uses within the function body. Note that there may be multiple such edges, as Figure 8.3 illustrates, although other figures in the chapter show only one edge to avoid clutter.

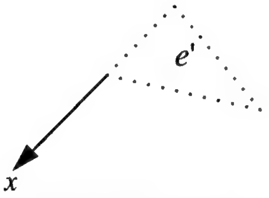
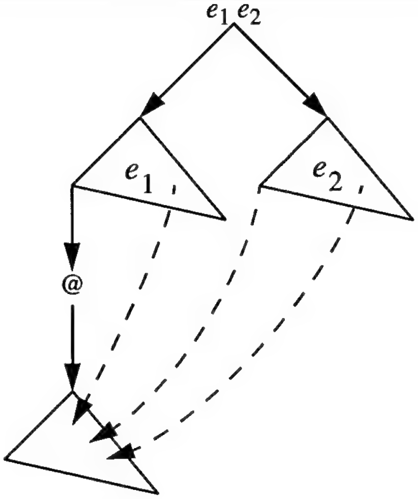
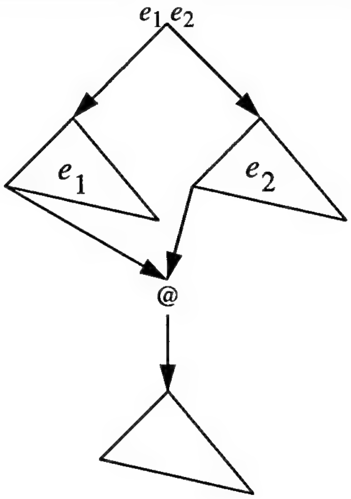
| Expression $e$: | $c$ or $\lambda x.e$ | $x$ |
|---|---|---|
| Graph $g$: | |  where $e'$ is the expression computing the value of $x$ |
| Expression $e$: | $e_1\ e_2$ | $e_1\ e_2$ |
| Graph $g$: |  where the last subgraph is for the body of the user-defined function (closure) to which $e_1$ evaluates |  where the last subgraph is for the application of the constant to which $e_1$ evaluates |

Figure 8.2: Illustration of computation graphs for the PSL model. The dashed lines represent possible dependencies.

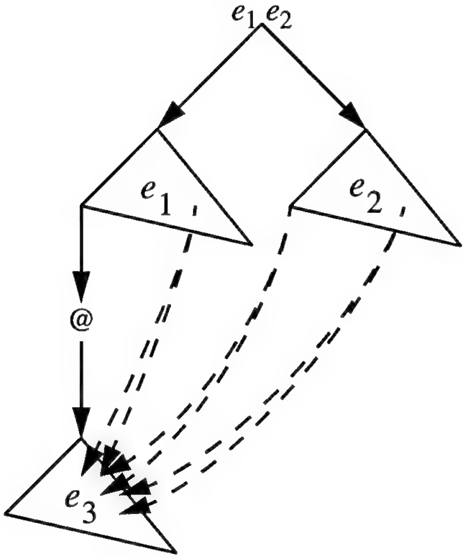


Figure 8.3: PSL computation graphs may have multiple edges from nodes.

- If the argument's value is a closure (or with the extended $\lambda$-calculus, also a data structure), it may communicate the name of another value being computed within the argument's subgraph. If that value is used in the function body, there is an edge from within the argument subgraph to its use, somewhere below the node linked to the argument's value. For example, if the argument constructs what represents a list, its value represents the first cons-cell. Edges would exist for each accessed element and cons-cell of the list, as illustrated in Figure 8.4.

- Similarly, the edge from the function to the function body may also communicate names of other values being computed within the function. These names are in the environment of the closure to which the function evaluates.

- Unlike PAL computation graphs, those of the PSL model are not compositional in terms of their subgraphs. This complicates the operators for defining graphs.

For an application expression, the node between the function and the function body is included only for consistency with the PAL model. It represents the application of the function value and a placeholder for its argument. It could be omitted with a resulting constant factor difference in the work and depth.[1]

[1] This extra node was not included in the previous version of this work [47]. Thus, the costs of this presentation are a constant factor greater than those of that earlier presentation.

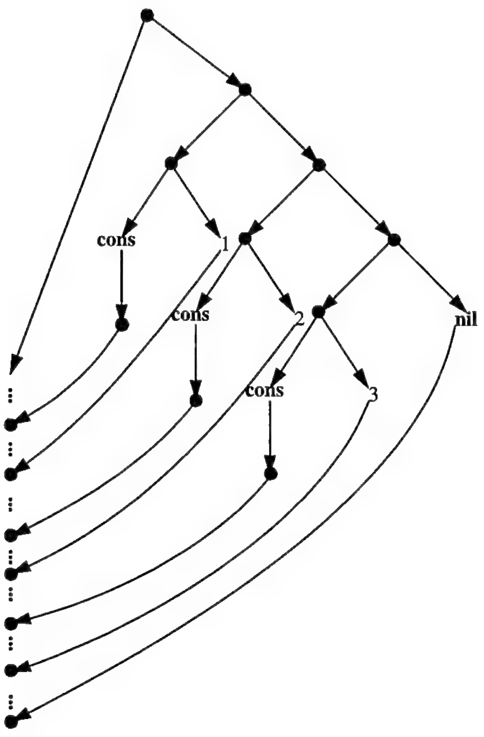


Figure 8.4: Illustration of PSL computation graph where the function branch accesses a list's elements in order and the argument branch creates the list. To access each element, the function must first access the cons-cell containing the element. The graph is simplified, with some nodes consolidated, but still representing constant amounts of computation.

The asymmetric nature of these graphs leads to a useful notion of threads, which is formally defined in the abstract machine. The thread evaluating an application expression

- spawns a new thread to evaluate the argument,
- evaluates the function, and
- evaluates the function body.

Recursively, the evaluation of both the function and function body generally use additional threads. For example in Figure 8.5, evaluating $(e_1\ e_2)\ e_3$, the initial thread spawns a new thread for $e_3$, then evaluates the inner application, spawning a new thread for $e_2$, and then evaluates $e_1$. The same thread then evaluates the function bodies $e'_1$ and $e'_{12}$. Although not shown, additional data edges may come into the graph for the inner application body $e'_1$ from outside the graph for the application $e_1\ e_2$. As in the following example, threads follow the leftmost control edges until encountering a parent thread or the thread simply ends.

**Example 8.1** *Consider the evaluation of the following expression:*

$$(\lambda x.1)\ ((\lambda y.e)\ 2).$$

*The computation graph for this expression is given in Figure 8.6. The leftmost edges including the root represent the main thread; the middle branch, another thread; and the right-most node, another. The evaluation of $e$ may use additional threads. No thread synchronizes with the main thread, but the right-most thread might need to synchronize with the evaluation of $e$.*

One general pattern of graphs possible arises from a consumer-producer relationship. In an application $(\lambda x.e_3)\ e_2$, execution of the trivial function is immediately followed by that of the function body $e_3$. This occurs in parallel with the execution of the argument $e_2$. Examining the graph from Figure 8.2, we see that the argument can act as a producer of some arbitrarily large data structure, and the body acts as a consumer of that data.

### PSL **computation graphs formalized**

As Figure 8.2 shows, graphs are always built with edges leading from the minimum, not the maximum, sink. Thus maximum sinks are not part of the formal definition of the computation graphs, but are only a descriptive tool. For example, no computation can simply "stall", delaying with no work, until all of a list is built, where the tail of the list is computed at the maximum sink of a subgraph. Instead, the computation would need to perform work such as counting down the list to delay until the list was built.

**Definition 8.1** *A* speculative (or PSL) computation graph *is a triple $(ns, nt, NE)$ of its source, minimum sink, and a mapping representing adjacency lists as before.*

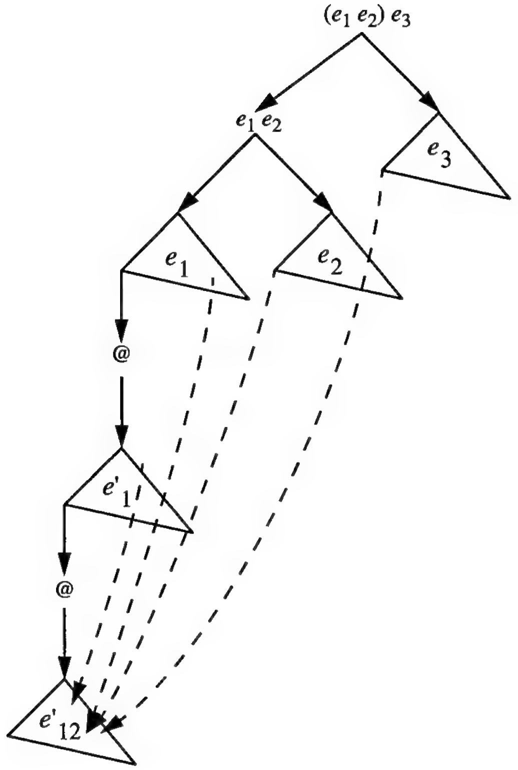


Figure 8.5: PSL computation graph of nested applications $(e_1\ e_2)\ e_3$, where $e_1$ evaluates to $\lambda x.e'_1$, and $e_1\ e_2$ evaluates to $\lambda y.e'_{12}$.

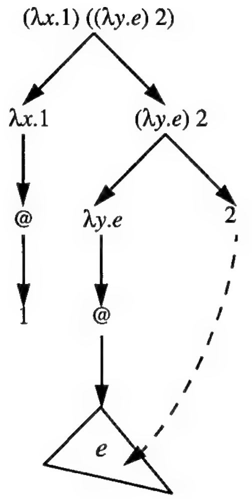


Figure 8.6: PSL computation graph for Example 8.2.

Figure 8.7 shows the definition of the combining operators for these computation graphs. Because of the cross links in the graph for an application, this definition is not compositional. Similarly, the minimum and maximum depths (the depths of the corresponding sink nodes) cannot be computed compositionally, although the work of a graph can be.

The major differences from the operators for PAL graphs are as follows:

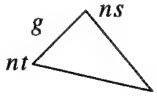
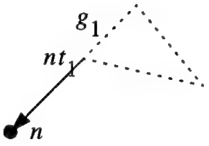
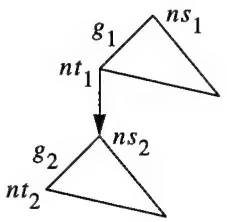
- The operator $\mathbf{1}\leftarrow g$ creates a new singleton node and also creates an edge to this node from the minimum sink of $g$. This represents the synchronization at a variable in a function body. Alone, this does not form a valid graph, but must be used in conjunction with the other operators. In particular, this is never the only edge to the node.

- A new parallel operator $g_1 \wedge g_2$ is introduced to fork graphs. Note that we can define a fork-and-join parallel operator by $g_1 \otimes g_2 = (g_1 \wedge g_2) \oplus (\mathbf{1}\leftarrow g_2)$.

- Both the serial and parallel operators use $\uplus$ to combine mappings representing sets of adjacency lists. The domains of their adjacency lists may overlap if they each contain an edge from the same node, which may occur if either graph joined by an operator includes a data edge created by $\mathbf{1}\leftarrow g$, for some $g$. Figure 8.8 illustrates one case where this arises. Combining adjacency lists with potentially overlapping domains uses $\uplus$.

### 8.1.2 Semantics

We now define the PSL model using a profiling semantics. Since the language is the same as before, its space under serial evaluation is the same, and we omit it from this profiling

| Graph $g$: | $\mathbf{1}$ | $\mathbf{1}\leftarrow g_1$ | $g_1 \oplus g_2$ |
|---|---|---|---|
| $(ns, nt, NE)$<br><br> | $(n, n, \cdot)$<br>unique $n$<br> <br> | $(n, n, [nt_1 \mapsto [n]])$<br>unique $n$<br><br> | $(ns_1, nt_2$<br>$(NE_1 \uplus NE_2)[nt_1 \mapsto [ns_2]])$<br><br> |
| $W(g)$: | 1 | 1 | $W(g_1) + W(g_2)$ |

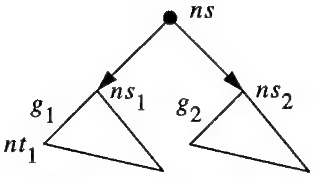
| Graph $g$: | $g_1 \wedge g_2$ |
|---|---|
| | $(ns, nt_1, (NE_1 \uplus NE_2)[ns \mapsto [ns_1, ns_2]])$<br>unique $ns$<br><br> |
| $W(g)$: | $W(g_1) + W(g_2) + 1$ |

Figure 8.7: The definition of combining operators for PSL computation graphs and work.

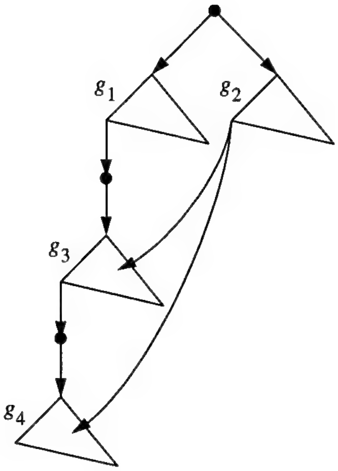


Figure 8.8: Illustration of a case where combined computation graphs share edges from the same node. This shows the graph $(g_1 \wedge g_2) \oplus \mathbf{1} \oplus g_3 \oplus \mathbf{1} \oplus g_4$, where each of $g_3$ and $g_4$ contains a subgraph $\mathbf{1}\!\leftarrow g_2$.

semantics. However, we still explicitly manage the memory via stores, since we use a store in the implementation as well.

A significant difference from the PAL semantics is needed to describe synchronization. In this model, environments maps each variable to both a value and the computation graph describing the evaluation to that value, as in Figure 8.9. This is used to describe when that value has been computed and can be looked up in the environment. The profiling semantics is then given by Definition 8.2.

**Definition 8.2** (PSL **profiling semantics**) *In the* PSL *model,* starting with the environment $\rho$ and store $\sigma$, the expression $e$ evaluates to value $v$ and the new store $\sigma'$ with compu-

| | | | | | |
|---|---|---|---|---|---|
| $l$ | $\in$ | *Locations* | | | |
| $v$ | $\in$ | *Values* | $::=$ | $c \mid l$ | |
| $sv$ | $\in$ | *StoreValues* | $::=$ | $\mathbf{cl}(\rho,x,e)$ | closure |
| $\rho$ | $\in$ | *Environments* | $=$ | $\mathit{Variables} \overset{fin}{\to} \mathit{Values} \times \mathit{Graphs}$ | |
| $\sigma$ | $\in$ | *Stores* | $=$ | $\mathit{Locations} \overset{fin}{\to} \mathit{StoreValues}$ | |

Figure 8.9: PSL run-time domains.

$$\rho, \sigma \vdash c \xrightarrow{\text{PSL}} c, \sigma; \mathbf{1} \qquad \text{(CONST)}$$

$$\rho, \sigma \vdash \lambda x.e \xrightarrow{\text{PSL}} l, \sigma[l \mapsto \mathbf{cl}(\rho,x,e)]; \mathbf{1} \quad \text{where } l \notin \sigma \qquad \text{(LAM)}$$

$$\frac{\rho(x) = v; g}{\rho, \sigma \vdash v \xrightarrow{\text{PSL}} v, \sigma; \mathbf{1}\!\leftarrow g} \qquad \text{(VAR)}$$

$$\frac{\begin{array}{c}\rho, \sigma \vdash e_1 \xrightarrow{\text{PSL}} l, \sigma_1; g_1 \qquad \rho, \sigma_1 \vdash e_2 \xrightarrow{\text{PSL}} v_2, \sigma_2; g_2 \\ \sigma_2(l) = \mathbf{cl}(\rho',x,e_3) \qquad \rho'[x \mapsto v_2; g_2], \sigma_2 \vdash e_3 \xrightarrow{\text{PSL}} v_3, \sigma_3; g_3\end{array}}{\rho, \sigma \vdash e_1\ e_2 \xrightarrow{\text{PSL}} v_3, \sigma_3; (g_1 \wedge g_2) \oplus \mathbf{1} \oplus g_3} \qquad \text{(APP)}$$

$$\frac{\rho, \sigma \vdash e_1 \xrightarrow{\text{PSL}} c, \sigma_1; g_1 \qquad \rho, \sigma_1 \vdash e_2 \xrightarrow{\text{PSL}} v_2, \sigma_2; g_2 \qquad \delta(\sigma_2, c, v_2) = v_3, \sigma_3; g_3}{\rho, \sigma \vdash e_1\ e_2 \xrightarrow{\text{PSL}} v_3, \sigma_2 \cup \sigma_3; (g_1 \wedge g_2) \oplus (\mathbf{1}\!\leftarrow g_2) \oplus g_3} \qquad \text{(APPC)}$$

Figure 8.10: The profiling semantics of the PSL model using the definition of $\delta$ in Figure 5.13.

tation graph $g$, *or*

$$\rho, \sigma \vdash e \xrightarrow{\text{PSL}} v, \sigma'; g,$$

*if it is derivable from the rules of Figure 8.10. The $\delta$ function for the application of constants is given in Figure 5.13.*

Aside from the inclusion of graphs in the environment, the PSL profiling semantics is much like that of the PAL semantics. The constant, abstraction, and constant application rules are the same. To see that for APPC, remember that $g_1 \otimes g_2 = (g_1 \wedge g_2) \oplus (\mathbf{1}\!\leftarrow g_2)$. The variable rule is basically the same, except that the singleton node adds an edge from the minimum sink of the graph computing the value. The application of a general function is similar, except that its graph has no synchronization point, as that occurs in the variable rule. Note that its cost $(g_1 \wedge g_2) \oplus \mathbf{1} \oplus g_3$ is equivalent to $((g_1 \oplus \mathbf{1}) \wedge g_2) \oplus g_3$ and $((g_1 \oplus \mathbf{1} \oplus g_3) \wedge g_2)$.

**Example 8.2** *As a small example of a* PSL *profiling semantics derivation, observe the evaluation of* $(\lambda x.x)$ $(\lambda y.1)$ $2$ *and compare to the* PAL *derivation and graph for the same expression, as given in Example 5.2. The derivation tree is the same as that for the* PAL *evaluation, except for the costs. Figure 8.11 shows the overall computation graph. The left spine including the root represents the main thread; the other two nodes are separate threads, only one of which synchronizes.*

### Comparisons to similar semantics

By using computation graphs as our costs, we have been able to simplify the semantics as compared to those by Roe [105, 106] and by the author and Blelloch [47]. Similar to here, they

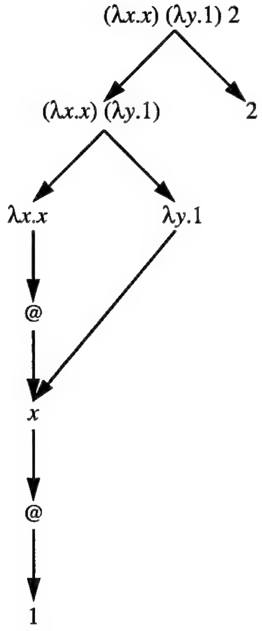


Figure 8.11: PSL computation graph for Example 8.2.

included depths in an environment to describe when values had been computed. But they also required the context of a judgment to contain a depth at which the evaluation begins, like threading clocks through the evaluation. Here we accomplish this result by building computation graphs which contain the same information in the connections.

Roe's semantics suffers from an additional complication resulting from his use of some serial expressions. Each expression results in two depths[2]: when the value becomes "available" (*i.e.*, the minimum depth) and when the evaluation has finished traversing the expression. This latter is just the minimum depth in our model, but to explain the difference in his, consider a pairing expression, $(e_1,e_2)$. The most natural rule for the PSL would express that the two subexpressions start evaluating at the same depth. But using its translation as in Section 5.5, the second subexpression starts at a constant lower depth than the first. This happens in Roe's model, despite there being a rule specifically for pairing expressions. Described operationally, first evaluation of the first component is started, then the second, then the pair is created where the component values eventually reside.

Another difference from Roe's semantics is that he tags every value with the depth at which it becomes available. This is a result of his inclusion of explicit data structures (cons-cells). These tags are subsumed here by

- tagging values in environments with their computation graphs (recall that the encoding of a data structure is a closure, which contains an environment), together with
- the evaluation judgment resulting in a value and its graph.

Flanagan and Felleisen [37] and Moreau [83, 84, 85] also provided semantics for speculative languages that were augmented with costs. Both used small-step contextual-style operational semantics and included continuations or escapes in the language. Moreau also included side-effects. Each described two measures of the work cost of evaluation: the total and the *mandatory*, *i.e.*, non-speculative, work. Note that in an expression $\textbf{let}\ x = e_1\ \textbf{in}\ e_2$, they considered $e_2$, not $e_1$, to be speculative since $e_1$ could abort, so that the result of $e_2$ would not be needed. Neither measured depth, or any related cost, even though both described parallel computation.

### 8.1.3 Recursion

This section outlines some issues about recursion in the PSL model and suggest one possible extension to incorporate explicit data recursion.

In the PAL model, it is unimportant for asymptotic bounds whether recursion was explicitly included or not. Here, that is not the case, but the problem is harder to describe. If we consider only recursive functions, the translation of Section 5.5 again introduces only constant work and depth overhead to each unrolling of a recursive function.

[2] Roe uses the word "time".

$$\frac{\begin{array}{c}\rho', \sigma \vdash e_1 \xrightarrow{\text{PSL}} v_1, \sigma_1; g_1 \\ \rho', \sigma_1[l \mapsto v'; g_1] \vdash e_2 \xrightarrow{\text{PSL}} v, \sigma_2; g_2\end{array}}{\rho, \sigma \vdash \mathbf{letrec}\ x = \mathbf{cons}\ e_1\ \mathbf{in}\ e_2 \xrightarrow{\text{PSL}} v, \sigma_2; \mathbf{1} \oplus g_2} \quad \text{where } l \notin \sigma,\ \rho' = \rho[x \mapsto l; g_1] \qquad \text{(LETREC-pair)}$$

Figure 8.12: Potential PSL rule for creating circular pairs. This assumes that the definition of expressions is suitably extended.

However, in a speculative semantics it might also make sense to create a subclass of recursive data structures—those that are circular. For example, consider the following recursive definition in an extended speculative language:

$$\mathbf{letrec}\ x = (e_1, x)\ \mathbf{in}\ e_2.$$

The semantics rule of Figure 8.12 would result in a circular pair. In this example, the definition returns a location for the pair value in constant work and depth and binds it to $x$ while the pair's components are still evaluating. The pair's second component returns having the value of the pair itself, circularly.

This form of a **letrec** expression can also be encoded in the basic $\lambda$-calculus, but its evaluation results in an infinitely long chain of pairs being created. Each pair in this chain is created by a separate thread in finite work and depth, but the overall computation never stops creating new threads for the rest of the chain. In full speculation without explicit recursion, the only way to terminate with circular data structures is to rewrite the program to delay and force the structures' components. In partial speculation without explicit recursion, any irrelevant threads should be eventually aborted, assuming the particular variant of partial speculation allows thread aborting to catch up with thread spawning, as discussed in Section 8.5.2.

## 8.2 Fully speculative intermediate model

This section defines the intermediate abstract machine for implementing the fully speculative version of the PSL model, PSLf. The basic structure of the P-CEK$^q_{\text{PSLf}}$ is similar to that of the P-CEK$^q_{\text{PAL}}$. It consists of a series of steps, each

- selecting at most $q$ active states from a stack, where we guarantee that these active states correspond to ready nodes in a graph traversal (*cf.* Section 6.1);
- executing computation and communication substeps on these states; and
- pushing any newly active states back onto the active state stack.

But while the basic structure is the same, the less structured synchronization pattern of speculation results in several differences.

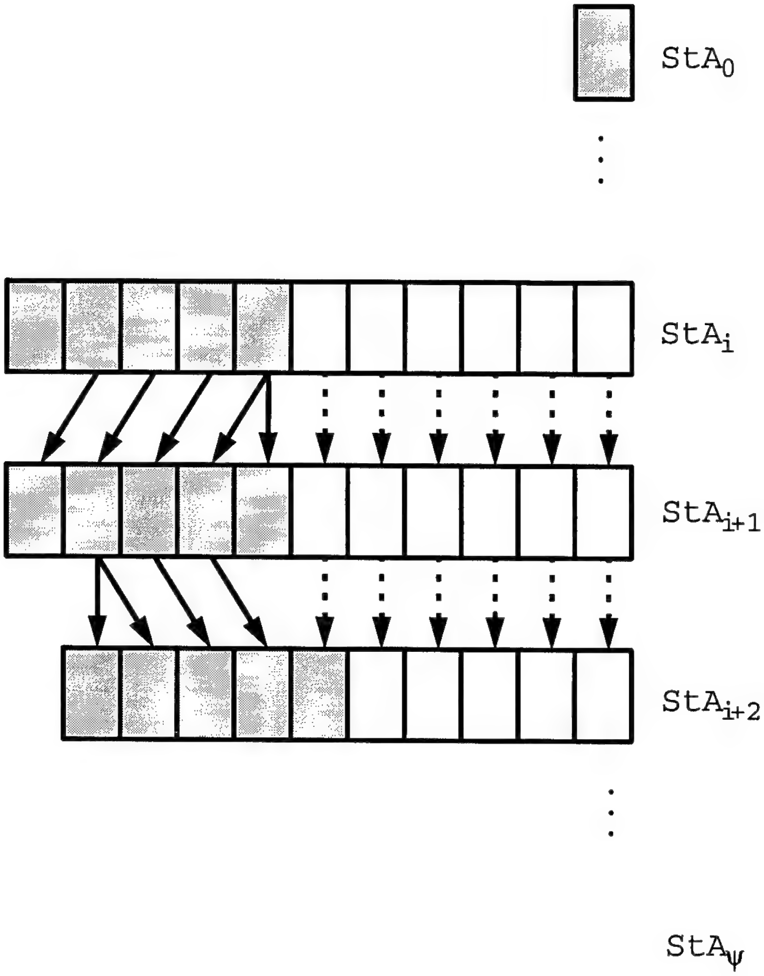


Figure 8.13: Illustration of P-CEK$^q_{\mathrm{PSLf}}$ active states during an evaluation. It starts with one active state representing the entire program and ends with no active states. The states are kept in a stack. At most $q$ states are selected each step. Here, $q = 5$ and these states are shaded. These can create zero or more new states (solid arrows). Unselected states are still active in the next step (dashed arrows).

Each step consists of one substep for computation and two substeps for communication and synchronization, as illustrated in Figure 8.14. The computation substep is much like that for the PAL model, except that instead of creating a new continuation for the argument of an application, we create a new thread for it. Note that the variable lookups are guaranteed not to block because active states correspond to ready nodes.[3] The two communication substeps manage sets of *suspended* states that have blocked. For each state that is finishing this step ($\mathsf{Fin}\langle v\ \tau\ \rangle$), the second substep *reactivates* the states that were blocked on the state. Reactivated states correspond to ready nodes since the value that they need is now available. Each of the states created by this step might block. These are suspended and placed in the appropriate sets in the final substep. Synchronization is required between the communication substeps to ensure that a thread's result value is found in the last substep if stored in the second substep.

We formalize the intuitive notion of a thread as a central data structure of the implementation. A thread represents a series of states over time computing the same value, as described previously. A *thread* $\tau$ is a pair of locations that contain the information common to these states:

- The first location contains either the resulting value of the thread or a marker Noval indicating the value has not been computed yet. Since a thread provides a pointer to its result value, environments now map variables to the threads evaluating them, rather than to their values. Of course, a more realistic implementation would also cache the value in the environment once computed, to avoid the extra indirection.

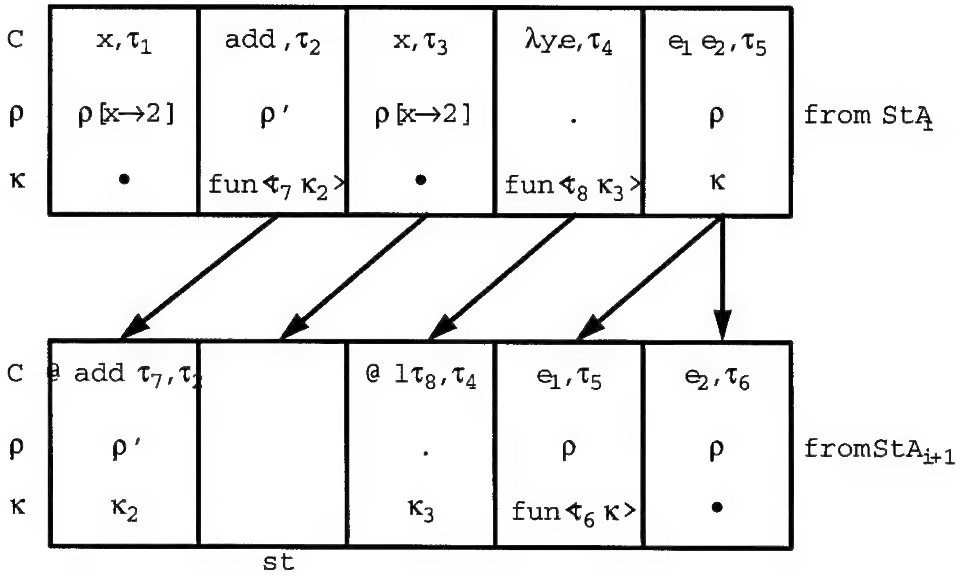
- The second location contains a set of states *suspended* on this thread, waiting for its value. When this thread finishes, these threads are reactivated so that they can continue with the value.

A thread's two components are selected with $\pi_1$ and $\pi_2$.

Each state records of which thread it is part by including the thread in its control string. This information is passed from state to state, for example, from application state to the state representing the function branch, and eventually to the state representing the function body. Only one state of a thread is accessible (*i.e.*, active or suspended) in the machine at any given time. Thus, while we describe the machine in terms of states for the sake of consistency, we could describe it equivalently in terms of threads.

Synchronization in an application happens on demand by the function body. Since it is out of the control of the argument, the argument can simply die when it is done, as long as the machine saves its result. By this we mean that the argument does not result in some intermediate state used for communication, as it does in the PAL model. Thus we do not

[3] In the previous presentation of this work [47], states blocked in the computation substep, rather than in a "pre-fetching"-like suspension substep. Thus active states did not correspond to ready nodes, and each state could be active on two steps—once when blocking and once when reactivated. As a result, the machine did not make a traversal of the computation graph, and the equivalence of the abstract machine and the profiling semantics was awkward.



$$
\begin{aligned}
\text{where}\quad \sigma_i(\pi_2(\tau_3)) &= [st] \\
l &\notin \sigma_i \\
\sigma_{i+1}(l) &= \mathbf{cl}(\cdot, y, e)
\end{aligned}
$$

Figure 8.14: Illustration of a P-CEK$^{q}_{\mathrm{PSLf}}$ step. States with constants and variables finish evaluation on this step and either reactivate any suspended threads or create a state for the appropriate function body. States with abstractions create a state to evaluate their bodies. Applications create two new states. New states may immediately block.

| | | | | | |
|---|---|---|---|---|---|
| $\tau$ | $\in$ | *Threads* | $::=$ | $(l, l)$ | |
| | | *ValueOpts* | $::=$ | $\mathsf{Noval} \mid \mathsf{Val}\ v$ | |
| $e$ | $\in$ | *Expressions* | $::=$ | $\ldots \mid @\ v\ \tau$ | application |
| $C$ | $\in$ | *Controls* | $::=$ | $(e, \tau)$ | |
| $\rho$ | $\in$ | *Environments* | $=$ | $\mathit{Variables} \xrightarrow{fin} \mathit{Threads}$ | |
| $\kappa$ | $\in$ | *Continuations* | $::=$ | $\bullet \mid$ | thread finishing |
| | | | | $\mathsf{fun}\langle \tau\ \kappa \rangle$ | function finishing |
| $st$ | $\in$ | *States* | $::=$ | $(C, \rho, \kappa)$ | |
| $St$ | $\in$ | *StateArrays* | $::=$ | $\vec{st}$ | |
| $I$ | $\in$ | *IntermediateStates* | $::=$ | $St \mid$ | new/reactivated states |
| | | | | $\mathsf{Fin}\langle v\ \tau \rangle$ | finishing state |
| $\sigma$ | $\in$ | *Stores* | $=$ | $\mathit{Locations} \xrightarrow{fin} (\mathit{StoreValues}+$<br>$\mathit{ValueOpts}+$<br>$\mathit{ThreadSets})$ | |

Figure 8.15: P-CEK$^{q}_{\text{PSLf}}$ domains. The ellipsis represents the expressions of Figure 4.1.

need a continuation for arguments. The function continuation is used simply to record the thread of its argument so that this information can be passed to the function body.

The initial state and thread of the computation uses a location $l_{res}$ for the thread's eventual result. When the machine finishes, the result value $v$ is in that location; thus the special state with **done** $v$ is not needed. So this machine ends when there are no active states left.

The above descriptions lead us to the definition of domains for the intermediate machine given in Figure 8.15, where the ellipsis represents the basic $\lambda$-calculus expressions of Figure 4.1.

**Definition 8.3 (P-CEK$^{q}_{\text{PSLf}}$ step)** *A step $i$ of the P-CEK$^{q}_{\text{PSLf}}$ machine, written*

$$StA_i, \sigma_i \overset{\text{PSLf},q}{\hookrightarrow} StA_{i+1}, \sigma_{i+1}; Q_i,$$

*is defined in Figure 6.5. It starts with a stack of active states $StA_i$ and a store $\sigma_i$ and produces a new stack and store for the next step. This step processes $Q_i$ states.*

**Definition 8.4 (P-CEK$^{q}_{\text{PSLf}}$ evaluation)** *In the P-CEK$^{q}_{\text{PSLf}}$ machine, the* evaluation of expression $e$ to value $v$, starting in the environment $\rho$ and store $\sigma_0$, ends with store $\sigma_\psi$ and processes $Q$ states, in $\psi$ steps using $s$ reachable space, *or*

$$\rho, \sigma_0 \vdash e \overset{\text{PSLf},q}{\Longrightarrow} v, \sigma_\psi; Q, \psi, s.$$

*For each of these $i \in \{0, \ldots, \psi - 1\}$ steps,*

$$StA_i, \sigma_i \overset{\mathrm{PSLf},q}{\hookrightarrow} StA_{i+1}, \sigma_{i+1}; Q_i,$$

*such that*

- *the machine starts with one active state and one thread for the whole program: $StA_0 = [((e, \tau), \rho, \bullet)]$, $\sigma_0 = [l_{res} \mapsto \mathsf{Noval}][l'_{res} \mapsto [\,]]$;*
- *the machine ends with zero active states and the result value: $StA_\psi = [\,]$, $\sigma_\psi(l_{res}) = v$; and*
- *the total number of states processed is $Q = \sum_{i=0}^{\psi-1} Q_i$.*

The computation substep's transition $\overset{\mathrm{PSL}}{\hookrightarrow}_{comp}$ corresponds to that of the P-CEK$^q_{\mathrm{PAL}}$ in that it performs a case analysis on the state's expression and generates up to two new states or a special intermediate state for use in the following substeps. The main differences are as follows:

- A variable lookup must get the value from the appropriate thread. A previous step's use of the third substep guarantees this value is available when requested.
- All expressions are tagged with a thread identifier. An application creates a new thread for the argument.

The step uses this transition in parallel for each of the selected active states.

The communication substeps finish and suspend states, respectively. For each intermediate state $\mathsf{Fin}\langle v\ \tau\rangle$, the second substep uses the $\overset{\mathrm{PSL}}{\hookrightarrow}_{react}$ transition to reactivate the states suspended on thread $\tau$ and store its result value $v$. For each newly created state, the third substep uses the $\overset{\mathrm{PSL}}{\hookrightarrow}_{block}$ transition to check whether it would block and if so, add it to the set of suspended states owned by the thread on which it would block. The blocked thread reactivates in a later instance of the second substep once the value is available. While Figure 8.16 describes the semantics sequentially for simplicity, the following section shows that the instances of this transition can be parallelized so that all these states suspend at once.

The step ends by adding the new states that haven't blocked and the reactivated states to the active states stack. In the P-CEK$^q_{\mathrm{PSLf}}$, the active states do not need to be treated as a stack—since states block and reactivate in arbitrary patterns, a stack does not obtain a $q$-DFT ordering of the computation graph. Maintaining them in a stack is one way to ensure determinacy. Obtaining the $q$-DFT ordering would require sorting the new and reactivated states based on their 1-DFT ordering and merging them into the non-selected active states, which cannot be done in the time bounds we show.

Since each transition represents constant work, the total work for the step is defined as $q'$. The space cost is the reachable space at the beginning of the step, which is within a

| $st$ | | | $I$ | | if/where |
|---|---|---|---|---|---|
| $((c,\quad \tau),\ -,\ \kappa)$ | $-$ | $\overset{\text{PSL}}{\hookrightarrow}_{comp}$ | $throw(c,\tau,\kappa)$ | $\cdot$ | |
| $((x,\quad \tau),\ \rho,\ \kappa)$ | $\sigma$ | $\overset{\text{PSL}}{\hookrightarrow}_{comp}$ | $throw(v,\tau,\kappa)$ | $\cdot$ | $\sigma(\pi_1(\rho(x))) = \mathsf{Val}\ v$ |
| $((\lambda x.e,\quad \tau),\ \rho,\ \kappa)$ | $\sigma$ | $\overset{\text{PSL}}{\hookrightarrow}_{comp}$ | $throw(l,\tau,\kappa)$ | $[l \mapsto \mathbf{cl}(\rho',x,e)]$ | $\rho' = restr(\rho, \lambda x.e),\ l \notin \sigma$ |
| $((e_1\ e_2,\quad \tau),\ \rho,\ \kappa)$ | $\sigma$ | $\overset{\text{PSL}}{\hookrightarrow}_{comp}$ | $[((e_1,\tau),\rho,\mathsf{fun}\langle\tau'\ \kappa\rangle),$<br>$((e_2,\tau'),\rho,\bullet)]$ | $[l \mapsto \mathsf{Noval},$<br>$l' \mapsto []]$ | $\tau' = (l,l'),\ l,l' \notin \sigma$ |
| $((@\ l\ \tau',\ \tau),\ \cdot,\ \kappa)$ | $\sigma$ | $\overset{\text{PSL}}{\hookrightarrow}_{comp}$ | $[((e,\tau),\rho[x \mapsto \tau'],\kappa)]$ | $\cdot$ | $\sigma(l) = \mathbf{cl}(\rho,x,e)$ |
| $((@\ c\ \tau',\ \tau),\ \cdot,\ \kappa)$ | $\sigma$ | $\overset{\text{PSL}}{\hookrightarrow}_{comp}$ | $throw(v',\tau,\kappa)$ | $\sigma'$ | $\sigma(\pi_1\tau') = \mathsf{Val}\ v,$<br>$\delta(\sigma,c,v) = v',\sigma'; -$ |

$$
\begin{array}{llcl}
\text{where} & throw(v,\ \tau,\ \bullet) & = & \mathsf{Fin}\langle v\ \tau\rangle \\
& throw(v,\ \tau,\ \mathsf{fun}\langle\tau'\ \kappa\rangle) & = & [((@\ v\ \tau',\tau),\cdot,\kappa)] \\
& & & \\
& restr(\ \rho,\ e) & = & \text{the environment } \rho \text{ restricted to the free variables in } e
\end{array}
$$

| $I$ | | | $St$ | |
|---|---|---|---|---|
| $\mathsf{Fin}\langle v\ \tau\rangle$ | $\sigma$ | $\overset{\text{PSL}}{\hookrightarrow}_{react}$ | $\sigma(\pi_2\tau)$ | $[\pi_1\tau \mapsto \mathsf{Val}\ v]$ |
| $\vec{st}$ | $-$ | $\overset{\text{PSL}}{\hookrightarrow}_{react}$ | $[]$ | $\cdot$ |

| $st$ | | | $St$ | | if/where |
|---|---|---|---|---|---|
| $((x,\quad \tau),\rho,\kappa)$ | $\sigma$ | $\overset{\text{PSL}}{\hookrightarrow}_{block}$ | $[]$ | $\sigma[\pi_2(\rho(x)) \mapsto [st] +\!\!+ \sigma(\pi_2(\rho(x)))]$ | $\sigma(\pi_1(\rho(x))) = \mathsf{Noval}$ |
| $((@\ c\ \tau',\tau),\cdot,\kappa)$ | $\sigma$ | $\overset{\text{PSL}}{\hookrightarrow}_{block}$ | $[]$ | $\sigma[\pi_2\tau' \mapsto [st] +\!\!+ \sigma(\pi_2\tau')]$ | $\sigma(\pi_1\tau') = \mathsf{Noval}$ |
| $st$ | $\sigma$ | $\overset{\text{PSL}}{\hookrightarrow}_{block}$ | $[st]$ | $\sigma$ | otherwise |

$$
StA,\sigma \overset{\text{PSLf},q}{\hookrightarrow} (+\!\!\!+\!\!\!+\ \vec{St}) +\!\!+ (+\!\!\!+\!\!\!+\ \vec{St'}) +\!\!+ [st_{q'},\ldots,st_{k-1}], \sigma'''; q', S_r(StA,\sigma)
$$

$$
\begin{array}{lrclll}
\text{if} & StA & = & [st_0,\ldots,st_{k-1}] & & \\
& q' & = & \min(q,k) & \text{select at most } q \text{ states per step} & \\
& st_i,\ \sigma & \overset{\text{PSL}}{\hookrightarrow}_{comp} & I_i,\quad \sigma_i & \text{for each } i \in \{0,\ldots,q'-1\} & \sigma' = \sigma \cup (\bigcup \vec{\sigma}) \\
& I_i,\ \ \sigma' & \overset{\text{PSL}}{\hookrightarrow}_{react} & St_i,\ \sigma'_i & \text{for each } i \in \{0,\ldots,q'-1\} & \sigma''_0 = \sigma' \cup (\bigcup \vec{\sigma'}) \\
& \vec{st'} & = & [st \mid st \in I_0,\ldots,I_{q'-1}] & \text{collect newly created states} & \\
& k & = & |\vec{st'}| & & \\
& st'_j,\ \sigma''_j & \overset{\text{PSL}}{\hookrightarrow}_{block} & St'_j,\ \sigma''_{j+1} & \text{“sequentially” for each } j \in \{0,\ldots,k-1\} & \sigma''' = \sigma''_k
\end{array}
$$

Figure 8.16: Definition of the P-CEK$^q_{\text{PSLf}}$ abstract machine step. Assume all new locations of the computation step are chosen or renamed to be distinct.

constant of that reachable at the end of the step. This would be defined in a similar manner to Definition 6.11, but also account for threads. But we omit the corresponding definition here, as we show no space bounds for the PSL implementations.

**Example 8.3** *As an example of execution in the P-CEK$^q_{\mathrm{PSLf}}$, Figure 8.17 shows the active states at the beginning of each step of evaluating the expression* $(\lambda x.\lambda y.x)\ ((\lambda z.z)\ (\mathbf{add}\ 1\ 2))$. *For lower values of q, the evaluation might take more steps, but it processes the same total number of states. The computation graph of the corresponding profiling semantics evaluation is shows for comparison in Figure 8.18, using the appropriate states' expressions as node labels. Observe that each of these executions is a greedy q-traversal of the graph. Also observe that the program's maximum depth is greater than its minimum depth, i.e., its value (a closure) is computed before the machine finishes.*

**Example 8.4** *If we apply the previous example program to another argument, the main thread must then wait for the computation of the value* 3, *as Figure 8.19 shows. The minimum and maximum sinks are then the same node.*

In these examples, the machine's executions are also $q$-DFTs because, for each thread, there as at most one thread that blocks on it. If more than one thread blocks on a given thread, the traversal depends on which blocks first. This results in a $q$-DFT if the reactivated threads are added to the active states in their 1-DFT order. Thus, the P-CEK$^q_{\mathrm{PSLf}}$ executes a $q$-DFT if threads blocking on a given thread block in their 1-DFT order.

**Comparisons to similar machines.** Moreau [84, 85] uses two similar, but more abstract, machines for speculative computation. The more detailed of the two is similar in form, as it is also based on the CEKS serial machine and consists of a series of steps transforming a collection of states and a store, such that each state contains a control string, environment, and continuation. It has three primary differences from ours:

- it includes side-effects and continuations;
- it does not explain how to schedule threads; and
- it uses one giant set of suspended threads, rather than a queue per thread.

**Pragmatics.** One way to reduce communication is to cache in the environment the results of fetching values from other threads. This is simple since the value of a thread never changes once computed.

We could reduce overhead by a constant factor by mutating the state of each thread, rather than creating a series of states over time. This would improve memory management on a real machine.

| $q \geq 4$ | | |
|---|---|---|
| **Step** $i$ | expressions in $StA_i$ | $q'$ |
| 0 | $\underline{(\lambda x.\lambda y.x)\ ((\lambda z.z)\ (\mathbf{add}\ 1\ 2))}$ | 1 |
| 1 | $\underline{\lambda x.\lambda y.x}$, $\underline{(\lambda z.z)\ (\mathbf{add}\ 1\ 2)}$ | 2 |
| 2 | $\underline{@\ l_0\ \tau_1}$, $\underline{\lambda z.z}$, $\underline{\mathbf{add}\ 1\ 2}$ | 3 |
| 3 | $\underline{\lambda y.x}$, $\underline{@\ l_1\ \tau_2}$, $\underline{\mathbf{add}\ 1}$, $\underline{2}$ | 4 |
| 4 | $\underline{\mathbf{add}}$, $\underline{1}$ | 2 |
| 5 | $\underline{@\ \mathbf{add}\ \tau_3}$ | 1 |
| 6 | $\underline{@\ \mathbf{add}_1\ \tau_4}$ | 1 |
| 7 | $\underline{z}$ | 1 |
| | **States processed:** | 15 |

| $q = 2$ | | |
|---|---|---|
| **Step** $i$ | expressions in $StA_i$ | $q'$ |
| 0 | $\underline{(\lambda x.\lambda y.x)\ ((\lambda z.z)\ (\mathbf{add}\ 1\ 2))}$ | 1 |
| 1 | $\underline{\lambda x.\lambda y.x}$, $\underline{(\lambda z.z)\ (\mathbf{add}\ 1\ 2)}$ | 2 |
| 2 | $\underline{@\ l_0\ \tau_1}$, $\underline{\lambda z.z}$, $\mathbf{add}\ 1\ 2$ | 2 |
| 3 | $\underline{\lambda y.x}$, $\underline{@\ l_1\ \tau_2}$, $\mathbf{add}\ 1\ 2$ | 2 |
| 4 | $\underline{\mathbf{add}\ 1\ 2}$ | 1 |
| 5 | $\underline{\mathbf{add}\ 1}$, $\underline{2}$ | 2 |
| 6 | $\underline{\mathbf{add}}$, $\underline{1}$ | 2 |
| 7 | $\underline{@\ \mathbf{add}\ \tau_3}$ | 1 |
| 8 | $\underline{@\ \mathbf{add}_1\ \tau_4}$ | 1 |
| 9 | $\underline{z}$ | 1 |
| | **States processed:** | 15 |

where $l_0$ contains $\mathbf{cl}(-,x,\lambda y.x)$,
$l_1$ contains $\mathbf{cl}(-,z,z)$, and
$\tau_1$ and $\tau_2$ are defined in Figure 8.18.

Figure 8.17: P-CEK$^q_{\text{PSLf}}$ evaluations for Example 8.3. The underlined expressions correspond to the selected states of each step.

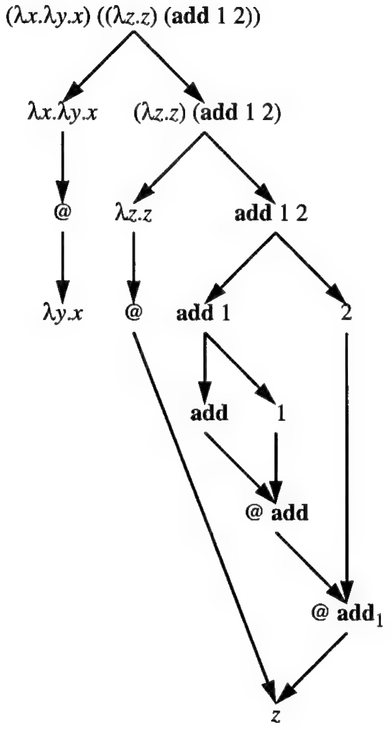


Figure 8.18: PSL computation graph for Example 8.3. The threads are numbered left-to-right, *i.e.*, the leftmost spine is $\tau_0$, the next leftmost is $\tau_1$, *etc.* Note that this is not the order of creation of the threads in this example.

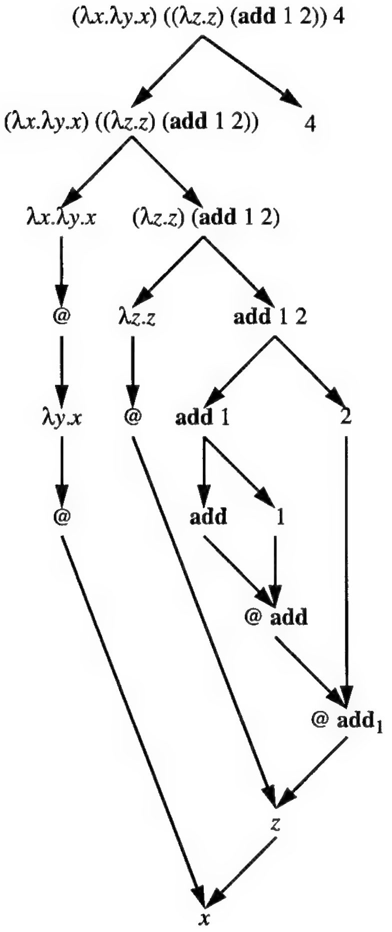


Figure 8.19: PSL computation graph for Example 8.4.

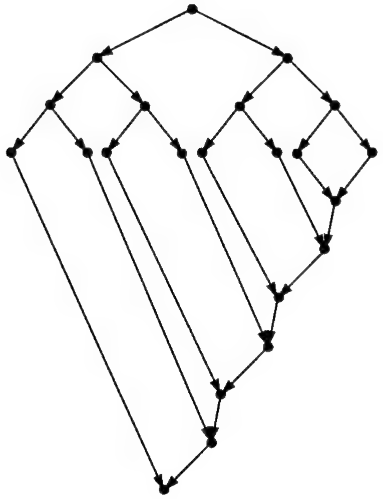


Figure 8.20: PSL computation graph dominated by a chain of dependencies.

The machine can build chains of suspended threads. For example, evaluation of the following expression:

$$(\lambda x.(\lambda y.(\lambda z.z)\ y)\ x)\ e$$

can have $z$ blocked waiting for the value of $y$, which is blocked on $x$, which is blocked on $e$. When $e$ finishes, it takes three steps for $x_3$ to get the value. The steps to follow this chain may dominate the computation. Consider the computation graph that Figure 8.20 displays. The first $\log_2 m$ levels spawn $m = 3$ threads which then form a chain of dependencies. The next $m - 1$ levels are needed to follow the chain. The following expression:

```
let x = let y = let z = 1
                in z
        in let z = y
           in z
in let y = let z = 1
           in z
   in let z = y
      in z
```

creates a graph similar to this, except that it requires $3 \log_2 m$ levels, $m = 3$, to build the chain (assuming **let** expressions are translated as in Section 5.5.2.

Successively halving the length of such chains would reduce the height of the computation graph to $O(\log m)$, but this probably cannot be implemented efficiently. This could be accomplished by having each thread $\tau_1$ that was blocked on a thread $\tau_2$ that was, in turn, blocked on a thread $\tau_3$ move itself from $\tau_2$'s suspended set onto $\tau_3$'s. We would also need to check that each of these threads has an empty continuation, so that they do not need to perform more computation when reactivated. But identifying such threads probably cannot be done efficiently, and the data structure used to implement suspended sets does not support an efficient delete operation.

### Equivalence of the language and intermediate models

This section relates the PSL profiling semantics to the P-CEK$^q_{\text{PSLf}}$ abstract machine. We show that the machine executes a greedy $q$-traversal of the computation graph. This provides a bound on the total states processed and steps taken by the machine. This also shows that the models compute the same result, although details of the extensional equivalence are omitted—they could be shown as for the equivalence of the PAL models. Since the machine does not execute a $q$-DFT, in general, we provide no bound on the total space. Thus we do not need to separately prove serial equivalence to obtain the serial space bound.

**Lemma 8.1 (P-CEK$^q_{\text{PSLf}}$ executes traversal)** *If $e$ evaluates in the profiling semantics:*

$$\cdot, \cdot \vdash e \overset{\text{PSL}}{\longrightarrow} v, \sigma; g,$$

*then it evaluates in the abstract machine:*

$$\cdot, \cdot \vdash e \overset{\text{PSLf},q}{\Longrightarrow} v, \sigma'; Q, \psi$$

*such that the machine executes a traversal of $g$. I.e.,*

- *the selected states and visited nodes correspond at each step, and*
- *the active states and ready nodes correspond at each step.*

*Proof Outline:* We prove this by induction on the steps of the machine. We could fully formalize this as in Lemma 6.1.

For brevity, we refer to states being visited or ready, rather than corresponding to nodes which are visited or ready, respectively. Clearly the initial state is ready, as it corresponds to the source of $g$.

Inductively, we need to show that any states added to the active states stack are ready on the next step—the non-selected states left in the stack remain ready. By a case analysis on the expression of each of the selected states, we see that the computation substep generates states corresponding to the graph.

Constants, variables, and abstractions finish immediately, thus this state corresponds to the unit graph specified for these expressions in the profiling semantics. Furthermore, a variable's lookup is the data dependency corresponding to the extra edge added by the profiling semantics.

Applications generate two new states to start evaluating the subexpressions. These correspond to the two parallel children of the application node. They are ready on the next step unless they are lookups of variables which do not have values yet, in which case it is suspended until ready. Both the function and argument evaluate inductively. Once the function finishes, the machine generates a state for @ $v$ $\tau$ that is immediately ready, corresponding to the node before the function body. When selected, the machine starts evaluating the function body, inductively (for a user-defined function) or via $\delta$ (for a constant function). Thus the evaluation corresponds to the graph. □

**Corollary 8.1** *If $e$ evaluates in the profiling semantics:*

$$\cdot, \cdot \vdash e \xrightarrow{\text{PSL}} v, \sigma; g,$$

*then it also evaluates in the abstract machine:*

$$\cdot, \cdot \vdash e \overset{\text{PSLf},q}{\Longrightarrow} v, \sigma'; Q, \psi$$

*such that the number of states processed by the machine equals the work of the profiling semantics: $Q = W(g)$.*

*Proof:* This follows from the one-to-one correspondence of active states processed and nodes in the graph. □

**Theorem 8.1** (PSLf **executes greedy** $q$**-traversal)** *If $e$ evaluates in the profiling semantics:*

$$\cdot, \cdot, \{\} \vdash e \xrightarrow{\text{PSLf}} v, \sigma; g, s,$$

*then it also evaluates in the abstract machine:*

$$\cdot, \cdot \vdash e \overset{\text{PSLf},q}{\Longrightarrow} v, \sigma'; Q, \psi, s'$$

*such that the machine executes a greedy $q$-traversal of $g$.*

*Proof:* This follows since the machine selects $\min(q, |StA|)$ nodes per step together with Lemma 8.1. □

**Corollary 8.2** *If $e$ evaluates in the profiling semantics:*

$$\cdot, \cdot \vdash e \xrightarrow{\text{PSL}} v, \sigma; g,$$

*then it also evaluates in the abstract machine:*

$$\cdot, \cdot \vdash e \overset{\text{PSLf},q}{\Longrightarrow} v, \sigma'; Q, \psi$$

*such that the number of steps in the machine is bounded by a function of the work and depth of the profiling semantics: $\psi \leq W(g)/q + D(g)$.*

*Proof:* This follows by Theorem 6.2. □

## 8.3 Representation of the sets of suspended states

The speculative implementation represents the active states the same as in the PAL implementation, with a SDGA. However, the sets of suspended states require a generalization of the single-threaded dynamically growing array (SDGA) used there. The SDGA operations assume that appending to multiple SDGAs does not need to be properly parallelized. However, in the speculative implementation, the machine must parallelize the suspension of threads onto multiple queues. To support that we introduce *multi-threaded dynamically growing arrays* (MDGAs). Two changes are made from the implementation of SDGAs:

- Uses of add-scan are replaced with uses of fetch-and-add (*cf.* Appendix A).
- We add an additional operation that pushes items onto multiple arrays in parallel.

We show that each step of a P-CEK$^{p\cdot TF(p)}_{\mathrm{PSLf}}$ machine can be implemented in $O(TF(p))$ amortized time, with high probability. Thus the cost bounds of the implementation are are parameterized by the cost of fetch-and-add. The amortization comes from how we grow the active state stack. Since we have a bound on the number of steps required by the machine, this allows us to bound the total running time for these machines.

In the new push operation, $m$ MDGAs of arbitrary size may need to grow at the same time, and we must parallelize the allocation and copying of all of the relevant data in these arrays. Each processor $i \in \{0,\ldots,p-1\}$ has states in an array $St_i$ to add to some MDGA $m_i$. Clearly there are at most $p$ MDGAs relevant to any given instance of this operation, so $m \leq p$. The state array of MDGA $j$ is labeled $St'_j$, for $j \in \{0,\ldots,m-1\}$. The operation is implemented as follows:

1. Compute (via a fetch-and-add) the number of states being added to each MDGA $j$ and the total number of states: $k_j =$ sum of $|St_i|$ such that $m_i = j$, $k = \sum_{i=0}^{p-1} |St_i|$.

   This requires $O(TF(p))$ time and $O(p)$ temporary space for the fetch-and-add operation.

2. Increase the size of MDGA arrays, where necessary, as Figure 8.21 illustrates.

   (a) Determine which MDGAs need larger arrays, and consider only these for the remainder of this step,

   This requires constant time to check the new length of each of the $m \leq p$ MDGAs, and $O(m)$ temporary space to store these lengths.

   (b) Create a single array $St$ such that each MDGA will use a sub-array of it. As before, each MDGA allocates twice as much space as its new total number of states. The length of $St$ is then the sum of the total space needed for each of the MDGAs and is computed with an add-reduce.

   This requires $O(TS(p))$ time, at most $O(k)$ control space, and $O(m)$ temporary space.

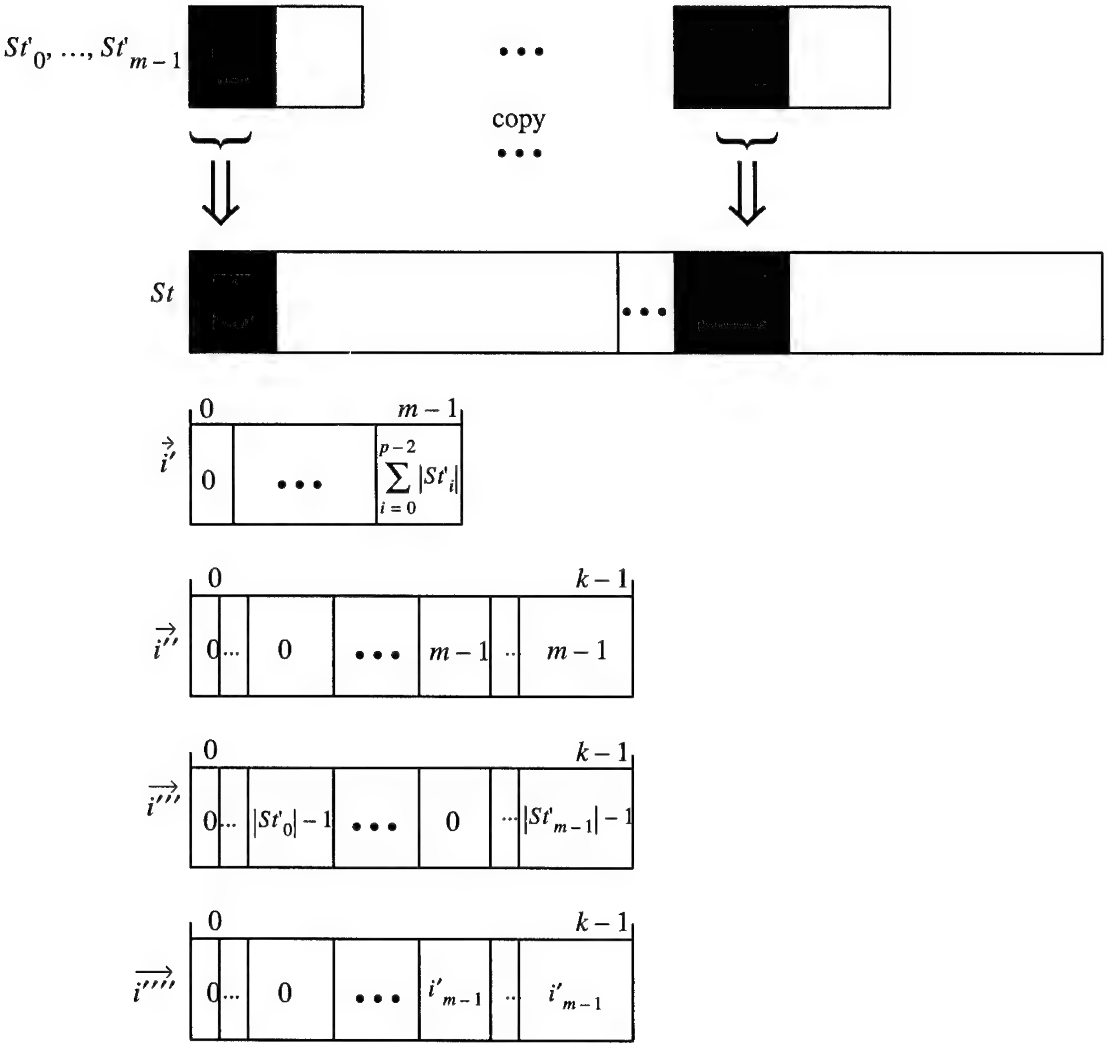


Figure 8.21: Step 2 of push operation on multi-threaded dynamically growing array (MDGA).

(c) Each MDGA computes (via a add-scan) the starting point within $St$ for its array, and stores this in $\vec{i'}$.

This requires $O(TS(p))$ time and $O(m)$ temporary space.

(d) For each location in array $St$ that receives an old state, record the source of its states. For example, the source for one location might be the 0th element of $St'_3$. Thus the sources are stored in an array $\vec{i''}$ of MDGA numbers (here, 3) and an array $\vec{i'''}$ of indices within the corresponding MDGA's states (here, 0). These is computed by a segmented distribute of $\vec{m}$ and a segmented index, respectively.

This requires $O(k/p + TS(p))$ time and $O(k)$ temporary space.

(e) For each location in array $St$ that receives an old state, record the destination of its states. The destinations are stored in an array $\vec{i''''}$ of offsets into $St$ marking where the new MDGA arrays start and in array $\vec{i'''}$. The former is computed by using $\vec{i''}$ to index into $\vec{i'}$.

This requires $O(k/p)$ time and $O(k)$ temporary space.

(f) Copy the current contents of these arrays into $St$. Each processor copies a proportional share of the array, using $\vec{i''}$ and $\vec{i'''}$ to index into the appropriate arrays $St'_j$.

For each MDGA $j$, the time for copying each of its $|St'_j|$ elements is counted against the time for initially writing the elements that will be written into the array until the next time it grows. There are at least $|St'_j|$ such elements, since the array doubles in size each time it grows. If the array doesn't grow again, the cost of this copy operation is counted instead against the initial writing of these elements. Thus, the time for copying data is at most twice that of initially writing data. This requires $O(k/p)$ amortized time and $O(k)$ space.

From now on, ignore the old arrays for these MDGAs and use the new ones.

Thus this step requires $O(k/p+TS(p))$ amortized time, $O(k)$ space, and $O(k)$ temporary space.

3. Move the new states into the MDGA arrays such that the load is evenly distributed among the processors, Figure 8.22 illustrates.

(a) For each location in the arrays $St'_j$ that receives a new state, record the source of its state. The sources are stored in an array $\vec{i'}$ of processor numbers and an array $\vec{i''}$ of indices within the corresponding processor's states. These are computed by a segmented distribute of the processor numbers and a segmented index of the lengths of $St_i$, respectively.

This requires $O(k/p + TS(p))$ time and $O(k)$ temporary space.

(b) For each location in the arrays $St'j$ that receives a new state, record this destination. The destinations are stored in an array $\vec{i'''}$ of MDGA numbers and an array

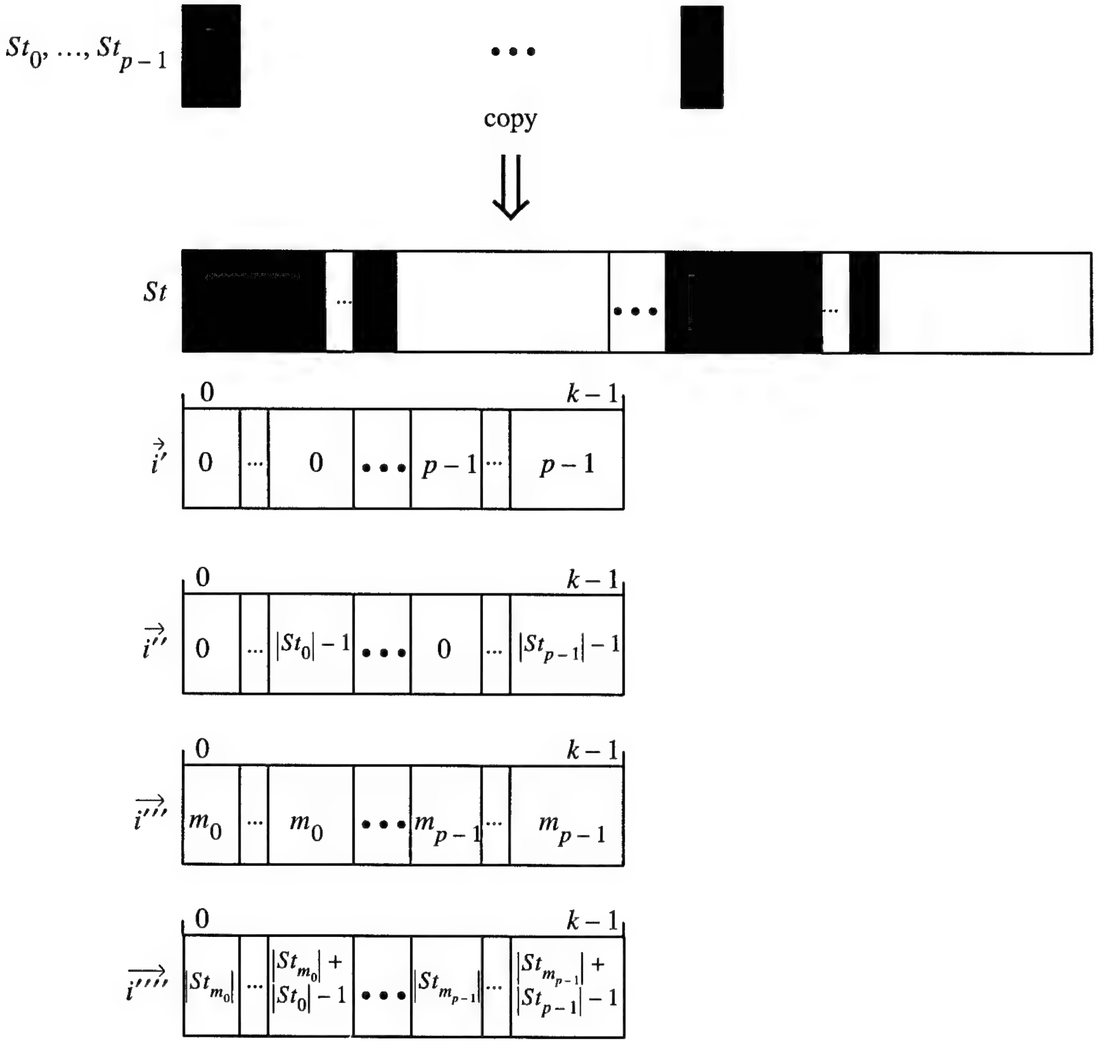


Figure 8.22: Step 3 of push operation on multi-threaded dynamically growing array (MDGA).

$\vec{i'''}$ of indices within the corresponding MDGA's states. The former computed by using $\vec{i'}$ to index into $\vec{m}$, and the latter by segment distributing the MDGA lengths and adding them to $\vec{i''}$.

This requires $O(k/p + TS(p))$ time and $O(k)$ temporary space.

(c) Copy the states into arrays $St'_j$. Each processor copies a proportional share of the data, using the sources and destinations just computed to control the indirect reads and writes, respectively.

This requires constant time per element, or $O(k/p)$ total time and no space.

Thus this step requires $O(k/p + TS(p))$ time and $O(k)$ temporary space.

So in total, this requires $O(k/p + TF(p))$ amortized time and $O(k)$ space for the data, plus $O(p)$ temporary space. This temporary space can be reused in each step.

## 8.4 Fully speculative machine models

Using the basic data structures just described, we now simulate the P-CEK$^q_{\text{PSLf}}$ on our machine models. First we examine the time required for each step of the P-CEK$^q_{\text{PSLf}}$, then total this for all steps.

**Theorem 8.2 (Cost of P-CEK$^q_{\text{PSLf}}$ step)** *Each P-CEK$^q_{\text{PSLf}}$ step can be simulated within $O(q/p + TF(p))$ amortized time on the p processor butterfly, hypercube, and PRAM machine models, w.h.p.*

*Proof:* Each processor is responsible for up to $q/p$ elements, *i.e.*, processor $i$ is responsible for the elements $i\lceil q'/p\rceil, \ldots, (i+1)\lceil q'/p\rceil - 1$, for $q' = \min(q, |StA|)$. We assume each processor knows its own processor number, so it can calculate a pointer to its section of the array.

The simulation of a step consists of the following:

1. locally evaluating the states ($\overset{\text{PSL}}{\hookrightarrow}_{comp}$), and synchronize all processors;

2. saving the result value, reactivating the queued states of all finishing states ($\overset{\text{PSL}}{\hookrightarrow}_{block}$), and synchronizing all processors;

3. suspending all states requesting to do so ($\overset{\text{PSL}}{\hookrightarrow}_{react}$); and

4. create a new active state stack for the next step.

We now show each of these is executed in the given bounds.

Locally evaluating the states requires the time it takes to process $k = \lceil q'/p\rceil$ states. The implementation of $\overset{\text{PSL}}{\hookrightarrow}_{comp}$ is straightforward and requires constant time for environment access and other operations. Thus the total time for locally evaluating the states on the machine models of interest is $O(k + TF(p))$, where $TF(p)$ provides an upper bound on any memory latency or space allocations.

In the second substep, each processor has up to $k$ states to finish. Each processor writes its states' results, then returns pointers to their suspended sets. This requires $O(k + TF(p))$ time, including memory latency.

In the third substep, each processor has up to $2k$ states that suspend. The states suspend with a single push operation, requiring $O(k + TF(p))$ amortized time.

The new active state stack is the appending of the newly created and reactivated states to the unselected original active states. There are at most $2q$ new states. On average, there are at most $2q$ reactivated states, since each state is blocked and later reactivated at most once. Thus we amortize over all steps, the number of states being added to the active states stack. We can push these elements onto the stack in $O(q/p + TF(p))$ amortized time. □

To account for memory latency in the butterfly and hypercube, and for the latency in the fetch-and-add operation for all three machines, we process $p \cdot TF(p)$ states on each step instead of just $p$. *I.e.*, we use a P-CEK$^{p\cdot TF(p)}_{\mathrm{PAL}}$ machine.

**Corollary 8.3** *Each step of the P-CEK$^{p\cdot TF(p)}_{\mathrm{PSLf}}$ machine can be simulated within $O(TF(p))$ amortized time on the p processor butterfly, hypercube, and PRAM machine models, w.h.p.*

**Corollary 8.4** *If e evaluates in the profiling semantics:*

$$\cdot, \cdot \vdash e \overset{\mathrm{PSL}}{\longrightarrow} v, \sigma; g,$$

*then its evaluation in the abstract machine,*

$$\cdot, \cdot \vdash e \overset{\mathrm{PSLf}, p\cdot TF(p)}{\Longrightarrow} v, \sigma'; Q, \psi,$$

*can be simulated within $O(W(g)/p + D(g)TF(p))$ amortized time on the p processor butterfly, hypercube, and PRAM machine models, w.h.p.*

## 8.5 Partially speculative implementations

A partially speculative implementation can abort and discard any irrelevant computation. We consider two definitions of relevance, varying in what program's result is treated.

1. If the result of the computation is only the semantic value obtained, *i.e.*, a constant or location, the definition is as follows:

   A node $n$ of a computation is *relevant* if there is a path from $n$ to the minimum sink $nt$ of the overall graph, *i.e.*, the final value depends on $n$.

2. If the result of the computation is either a constant or the entire data structure referenced from a location, the definition is as follows:

   A node $n$ of a computation is ***relevant*** if there is a path from $n$ to the minimum sink $nt$ of the overall graph, *i.e.*, the final value depends on $n$, or if $n$ is reachable from the value computed in $nt$, *i.e.*, it is part of the final result data structure.

Note that both of these definitions are stronger than saying that an application's argument is relevant if its value is used in its function body, since that function body (or that part of it) might not be relevant.

The simplest appropriate modification to the fully speculative implementation is to end on the first iteration that the main thread is done, *i.e.*, when $l_{res}$ has a value. But since the fully speculative implementation does not maintain the states in any particular order, we might be unlucky and schedule all irrelevant computation before the relevant computation. Thus it might make sense to prioritize computations to minimize the amount of irrelevant computation. This might also allow us to detect and discard irrelevant threads during evaluation.

Sections 8.5.1 and 8.5.2 discuss some strategies and implementations for prioritizing and aborting threads. Then Section 8.5.3 discusses the benefits of partial speculation.

### 8.5.1 Prioritizing threads

On each step, we select the (up to) $q$ active states with the highest priority. The goal of prioritizing threads is to minimize the number of irrelevant states used during a computation, so as to reduce the cost of the computation. Since we do not know whether a thread is relevant or not until the computation is done, any priority scheme is either very restricted or based on heuristics. Furthermore, more involved prioritization methods introduce more complicated data structures to store the active states. The cost of prioritizing threads has the potential for overwhelming the cost of evaluation.

The most basic priority scheme distinguishes *necessary* threads, those known to be relevant, from *speculative* threads, those not yet known to be relevant or irrelevant. Necessary threads are given higher priority than speculative threads. The initial thread is immediately necessary, as is any thread spawned from a necessary thread by APPC, or any thread that a necessary thread blocks on. To implement this scheme, we can use two active state stacks, one for each priority, with only a constant factor of overhead (*e.g.*, each newly created or reactivated state checks to which of the two stacks it should be added). This priority scheme was proposed by Baker and Hewitt [6].

More general priority schemes can be based on the distinguishing degrees of "speculativeness":

- Threads created by APPC have the same priority as the original parent thread, because they are relevant (or irrelevant) if and only if the original thread is.

- Threads created by APP are more speculative than the original thread, because the original parent thread must be used to communicate this one's name for it to be used.

The prioritization used by Partridge [94, 93] is an example of this. If we assume that each speculative child has equal probability to be relevant, then the active states should be kept as a tree, where each node represents active states of equal priority, and each edge represents a speculative child relationship. Selecting the highest priority threads is removing them from the top of the tree, which seems unlikely to be efficient. But adding new threads can be easily done by adding them in the appropriate places of the tree. Adding reactivated threads is also easy if we remember where they would have been placed when they blocked. Blocking also requires updating the priority of the thread blocked on to the higher of its current priority and the blocking thread's priority, but comparing two priorities is also unlikely to be faster than logarithmic in the depth of the priority tree.

Further generalizing the scheme to allow arbitrary probabilities of relevance would likely be even more inefficient. Maintaining the accurate thread ordering can dominate the cost of computation since it can involve touching many additional threads per step. As a simple example, consider storing the threads in order of relevance in an array. Inserting threads involves a sorted merge operation, requiring work linear in the number of currently active threads.

### 8.5.2 Aborting threads

To distinguish *unnecessary* threads, those known to be irrelevant, requires a form of garbage collection on threads. For example, consider the evaluation of an application $e_1$ $e_2$ resulting in a closure. Even if the function body does not use the argument $e_2$, the closure can contain a reference to the argument which is then used by the enclosing context. Only following all the relevant pointers can tell us which threads are no longer accessible, and thus unnecessary.

Several methods of garbage collection of processes has been previously described. Baker and Hewitt [6] and Hudak and Keller [54] used a mark-and-sweep approach, which is not asymptotically efficient since it traverses pointers too many times. Grit and Page [48] and Partridge [94, 93] used a reference counting approach which can be efficient if we don't spend too much effort garbage collecting on each step. We discuss this option in more detail.

For each thread we maintain a count of the references to this thread from environments. The count is one when the thread is created. Counts are incremented when environments are extended, creating a new copy of the environment.[4] And the appropriate counts are decremented when environments are restricted or threads finish.

When a state is selected at the beginning of a step, we check the count of its thread. If its count is zero, we abort this thread, including decrementing all counts of threads reachable from its environment. But these environments contains values, which in turn contain environments, *etc.*, and we cannot efficiently traverse the entire environment and decrement

[4] If environments are partially shared to minimize space as described in Section 6.2, the counts should appropriately reflect the sharing.

all these counts at once. So, we use a queue (implemented with a SDGA, but with queue operations) of environment bindings, decrementing $k \cdot q$ reference counts each step, for some constant $k$.

We can augment this scheme further by observing that all threads spawned by an irrelevant thread (*i.e.*, its *children*) are themselves irrelevant. When a thread is aborted, we also abort its children. To implement this, each thread also keeps a set of pointers to its children. When aborting a thread, the machine sets the counts of its descendants to zero and aborts them. While a given thread can have many more than $q$ descendants, the machine can amortize the cost of aborting them over all the steps, accounting this cost against the cost of creating the threads. Alternatively, we can use a queue (implemented with a SDGA[5]) of threads to abort, and abort $k \cdot q$ of these on each step, adding the children of any aborted thread onto the queue.

Reference counting is asymptotically efficient since we only need to change counts for threads that the machine is touching anyway. Thus it involves only a constant factor overhead.

The standard problem with reference counting in garbage collection is accounting for recursive data. Here it works with recursive functions because a recursive closure $\mathbf{cl}(\rho,x,y,e)$ would not be represented recursively (*cf.* the PAL' model). The key to that representation is explicitly naming the closure $x$ and unrolling the recursion only when necessary. The same technique could be applied to circular data structures. *E.g.*, the semantics suggested in Section 8.1.3 could be altered so that it created a named pair value, similar to the named closure, that is unrolled when applied to the selectors **fst** or **snd**.[6]

### 8.5.3 Cost benefits of partial speculation

When a priority scheme schedules computation well, for some computations it can greatly reduce the number of states processed or steps an evaluation. Clearly any partially speculative implementation should quickly detect that the potentially large computation $e$ in the expression $(\lambda x.1)\ e$ can be aborted.

Consider the subgraph $g_r$ consisting of only the relevant nodes of a computation graph $g$. Note that a serial call-by-need implementation executes a 1-DFT of $g_r$ and thus requires $W(g_r)$ time to evaluate the computation. Unlike the PSLf, any partially speculative abstract machine correctly prioritizing necessary computation processes at most $q \cdot W(g_r)$ states and $W(g_r)$ steps, since at least one relevant node is traversed on each step. *This implies that it terminates if the call-by-need implementation does.* But we conjecture that it is possible to construct, for any priority scheme, example computation graphs that the priority scheme does not significantly parallelize even if its relevant subgraph has significant parallelism. *I.e.*, no

[5] Grit and Page used a less efficient binary tree data structure for this.

[6] There are two alternatives as to what the "name" would be in this pair value. If we use lazy pairing, so that the pair component expressions are evaluated only as needed, the "names" are variables. Then the named value pair also includes an environment to perform the unrolling efficiently, *i.e.*, without substitution. If we retain eager pairing, the "names" need to be a new construct, and the named value pair includes a mapping from these names to values for unrolling efficiently.

priority scheme can correctly guess sufficiently often which speculative computation becomes necessary.

Garbage collecting threads obviously benefits the space cost of evaluation. But it is unclear whether it can improve the worst case asymptotic space bounds of an evaluation strategy. Consider an irrelevant subcomputation that generates many threads before the main thread of this subcomputation is known to be irrelevant. As the machine aborts these threads, their descendants, since they are not yet known to be irrelevant, can generate new irrelevant threads. The aborting of threads will eventually catch up with the spawning of new irrelevant threads if the machine does both of the following:

- It aborts more threads per step than it creates, *e.g.*, it aborts up to $k \cdot q$ threads per step, for some constant $k > 1$.

- It aborts on at least two levels of the graph each step. *E.g.*, its aborts some threads, which adds those threads' children to the abort queue and then aborts some of these children.

Without the latter condition, spawning could always be a level ahead of aborting, as in Grit and Page's description.

# Chapter 9

# Basic data-parallel models

This chapter presents a model of nested data-parallelism. Here we add sequences and parallel operations on sequences to the extended $\lambda$-calculus.[1] This models the core of the NESL language. As Figure 9.1 shows, the NESL model adds several constant functions on sequences and a sequence-based expression to the extended $\lambda$-calculus (Figure 4.3). These constants are described in the glossary of Appendix A. The "for-each" expression $\{e' : x \textbf{ in } e\}$ first evaluates $e$, which should result in a sequence. Then for each binding of $x$ to an element of this sequence, it evaluates $e'$. Each of these evaluations in done in parallel, and all of these synchronize before the rest of the computation continues. Only these new operations parallelize—applications and pairs do not, unlike the previous models.

| | | | | | |
|---|---|---|---|---|---|
| $c$ | $\in$ | *Constants* | $::=$ | $\ldots \mid$ **elt** $\mid \# \mid$ | sequence observers |
| | | | | **index** $\mid$ **dist** $\mid ++ \mid$ | sequence constructors |
| | | | | **pack** $\mid$ **put** | sequence mutators |
| | | | | **addscan** $\mid$ **maxscan** $\mid$ | sequence scans |
| $e$ | $\in$ | *Expressions* | $::=$ | $\ldots \mid \{e' : x \textbf{ in } e\}$ | for-each |

Figure 9.1: NESL expressions. The ellipses represent the constants and expressions of Figure 4.3.

For brevity, we do not include explicit constant sequences as expressions. *E.g.*, the sequence $[e_0, \ldots, e_{k-1}]$ can be encoded so its components evaluate in parallel:

$$\{\textbf{if eq } (x,0) \textbf{ then } e_0 \textbf{ else } \ldots : x \textbf{ in index } k\}$$

or in serial:

$$++ \; (\textbf{dist } (e_0,1), \ldots)$$

[1] As in the PAL' model, translating pairs, conditionals, *etc.*, to the basic $\lambda$-calculus would introduce only constant overhead in work, depth, and space, but here we include these language constructs for convenience.

Sections 9.1 and 9.2 define the computation graphs and profiling semantics, respectively, for the NESL model. Then Section 9.3 translates NESL to the ArrL model to make allocation more explicit, simplifying further implementation. Finally, Sections 9.4 and 9.5 implement the machine in an abstract machine and then in machine models, respectively, proving the time and space cost mappings of each stage of the implementation.

## 9.1 Computation graphs

Like the PAL model, the NESL model uses only series-parallel graphs, which allows us to use the same techniques as before to prove both time and space bounds on our implementation. But here, the unbounded parallelism of the for-each expression and other sequence constructs results in graph nodes of unbounded fan-out and fan-in. There are two main differences from those of the PAL model:

- Application and pairing are serialized, not parallelized.
- The for-each expression allows unbounded parallelism (as do some constant applications, to be shown). Its graph also contains $m$ "extra" parallelized nodes, where $m$ is the number of parallel branches of the body to execute. This ensures that we perform $m$ work before we allocation $m$ space for the result of the for-each.

As before, the computation graphs are formally defined in the profiling semantics. They use essentially the same set of combining operators as for PAL computation graphs, except that here we generalize the parallel operator, as Figure 9.3 shows.

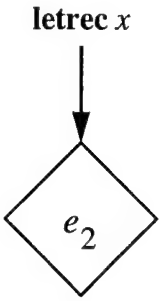
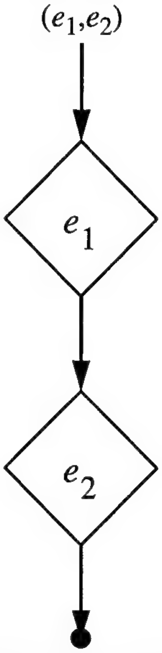
## 9.2 Profiling semantics

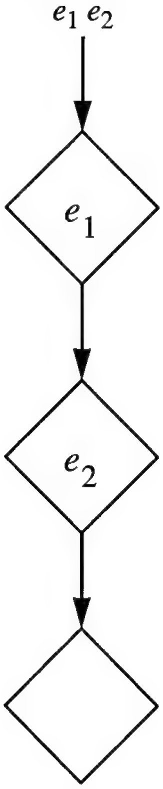
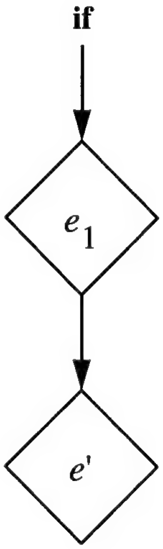
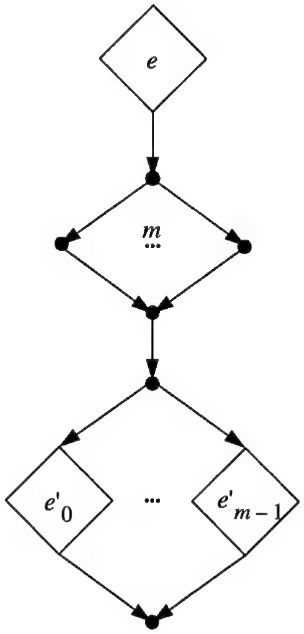
The profiling semantics for this model is essentially an extension of that of the PAL model, except that it serializes applications. A key extension is the addition of sequences as store values. Since we allow nested data-parallelism, sequences can contain locations, *e.g.*, to other sequences.

**Definition 9.1** (NESL **profiling semantics**) *In the* NESL *model,* starting with the environment $\rho$, store $\sigma$, and roots $R$, the expression $e$ evaluates to value $v$ and new store $\sigma'$ with computation graph $g$ and $s$ reachable space, *or*

$$\rho, \sigma, R \vdash e \xrightarrow{\text{NESL}} v, \sigma'; g, s,$$

*if it is derivable from the rules of Figure 9.5. The $\delta$ function for the application of constants is given in Figure 9.6. Additional functions for defining the space of a computation are given in Figure 9.7.*

| Expression $e$: | $c$, $x$, or $\lambda x.e$ | **letrec** $x$ $y = e_1$ **in** $e_2$ | $(e_1, e_2)$ |
|---|---|---|---|
| Graph $g$: | ● |  |  |

| Expression $e$: | $e_1$ $e_2$ | **if** $e_1$ **then** $e_2$ **else** $e_3$ | $\{e' : x$ **in** $e\}$ |
|---|---|---|---|
| Graph $g$: |  <br>where the last subgraph is that for either the body of the user-defined function |  <br>where $e'$ is either $e_2$ or $e_3$ depending on the value of $e_1$ |  <br>where there are $m$ (the length of the value of $e$) evaluations of $e'$ |

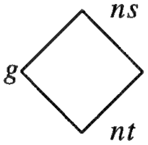
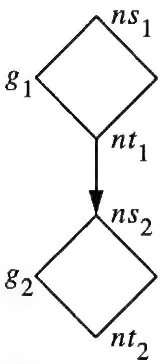
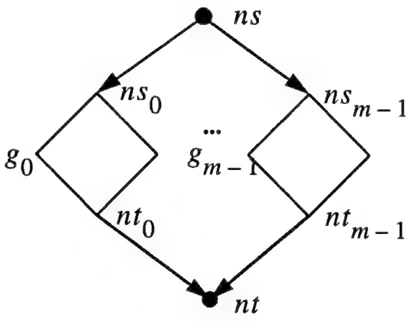
| Graph $g$: | $\mathbf{1}$ | $g_1 \oplus g_2$ | $\bigotimes_{i=0}^{m-1} g_i$ |
|---|---|---|---|
| $(ns, nt, NE)$ | $(n, n, \cdot)$<br>unique $n$ | $(ns_1, nt_2,$<br>$(NE_1 \cup NE_2)$<br>$[nt_1 \mapsto [ns_2]])$ | $(ns, nt,$<br>$(\bigcup_{i=0}^{m-1} NE_i[nt_i \mapsto [nt]])[ns \mapsto \vec{ns}])$<br>unique $ns$ and $nt$ |
|  | |  |  $ns$<br>$ns_0$<br>$ns_{m-1}$<br>$g_0$<br>...<br>$g_{m-1}$<br>$nt_0$<br>$nt_{m-1}$<br>$nt$ |
| $W(g)$: | 1 | $W(g_1) + W(g_2)$ | $2 + \sum_{i=0}^{m-1} W(g_i)$ |
| $D(g)$: | 1 | $D(g_1) + D(g_2) + 1$ | $2 + \max_{i=0}^{m-1} D(g_i)$ |

Figure 9.3: The definition of combining operators for NESL computation graphs, work, and depth.

$$\begin{array}{lcllll}
l & \in & \mathit{Locations} & & & \\
v & \in & \mathit{Values} & ::= & c \mid l & \\
sv & \in & \mathit{StoreValues} & ::= & \mathbf{cl}(\rho,x,y,e) \mid & \text{closure} \\
 & & & & \langle v_1,v_2 \rangle \mid & \text{pair} \\
 & & & & \vec{v} & \text{sequence} \\
\rho & \in & \mathit{Environments} & = & \mathit{Variables} \overset{fin}{\rightarrow} \mathit{Values} & \\
\sigma & \in & \mathit{Stores} & = & \mathit{Locations} \overset{fin}{\rightarrow} \mathit{StoreValues} & \\
R & \in & \mathit{Roots} & = & \mathit{ValueSets} &
\end{array}$$

Figure 9.4: NESL run-time domains.

$$\rho, \sigma, R \vdash c \xrightarrow{\text{NESL}} c, \sigma; \mathbf{1}, S(R, \sigma) \qquad \text{(CONST)}$$

$$\rho, \sigma, R \vdash x \xrightarrow{\text{NESL}} \rho(x), \sigma; \mathbf{1}, S(R \cup \{\rho(x)\}, \sigma) \qquad \text{(VAR)}$$

$$\rho, \sigma, R \vdash \lambda x.e \xrightarrow{\text{NESL}} l, \sigma'; \mathbf{1}, S(R \cup \{l\}, \sigma') \quad \text{where } \sigma' = \sigma[l \mapsto \mathbf{cl}(\rho,\_,x,e)],\ l \notin dom(\sigma) \qquad \text{(ABSTR)}$$

$$\frac{\begin{array}{c} \rho, \sigma, R \cup \rho(FV(e_2)) \vdash e_1 \xrightarrow{\text{NESL}} l, \sigma_1; g_1, s_1 \qquad \rho, \sigma_1, R \cup \{l\} \vdash e_2 \xrightarrow{\text{NESL}} v_2, \sigma_2; g_2, s_2 \\ \sigma_2(l) = \mathbf{cl}(\rho',x,y,e_3) \qquad \rho'[x \mapsto l][y \mapsto v_2], \sigma_2, R \vdash e_3 \xrightarrow{\text{NESL}} v, \sigma_3; g, s \end{array}}{\rho, \sigma, R \vdash e_1\ e_2 \xrightarrow{\text{NESL}} v, \sigma_3; \mathbf{1} \oplus g_1 \oplus g_2 \oplus \mathbf{1} \oplus g, \max(s_1 + 1, s_2 + 1, s)} \qquad \text{(APP)}$$

$$\frac{\begin{array}{c} \rho, \sigma, R \cup \rho(FV(e_2)) \vdash e_1 \xrightarrow{\text{NESL}} c, \sigma_1; g_1, s_1 \qquad \rho, \sigma_1, R \vdash e_2 \xrightarrow{\text{NESL}} v_2, \sigma_2; g_2, s_2 \\ \delta(c, v_2, \sigma_2) = v_3, \sigma_3; g_3 \end{array}}{\rho, \sigma, R \vdash e_1\ e_2 \xrightarrow{\text{NESL}} v_3, \sigma_2 \cup \sigma_3; \mathbf{1} \oplus g_1 \oplus g_2 \oplus g_3, \max(s_1 + 1, s_2 + 1, S(R \cup \{v\}, \sigma_3))} \qquad \text{(APPC)}$$

$$\frac{\begin{array}{c} \rho, \sigma, R \cup \rho(FV(e_2)) \vdash e_1 \xrightarrow{\text{NESL}} v_1, \sigma_1; g_1, s_1 \\ \rho, \sigma_1, R \cup \{v_1\} \vdash e_2 \xrightarrow{\text{NESL}} v_2, \sigma_2; g_2, s_2 \end{array}}{\rho, \sigma, R \vdash (e_1,e_2) \xrightarrow{\text{NESL}} l, \sigma_2[l \mapsto \langle v_1,v_2 \rangle]; \mathbf{1} \oplus g_1 \oplus g_2 \oplus \mathbf{1}, \max(s_1 + 1, s_2)} \quad \text{where } l \notin \sigma \qquad \text{(PAIR)}$$

$$\frac{\rho[x \mapsto l], \sigma[l \mapsto \mathbf{cl}(\rho,x,y,e_1)], R \vdash e_2 \xrightarrow{\text{NESL}} v, \sigma'; g_2, s}{\rho, \sigma, R \vdash \mathbf{letrec}\ x\ y = e_1\ \mathbf{in}\ e_2 \xrightarrow{\text{NESL}} v, \sigma'; \mathbf{1} \oplus g_2, s + 1} \quad \text{where } l \notin \sigma \qquad \text{(LETREC)}$$

$$\frac{\begin{array}{c} \rho, \sigma, R \cup \rho(FV(e_2)) \cup \rho(FV(e_3)) \vdash e_1 \xrightarrow{\text{NESL}} \mathbf{true}, \sigma_1; g_1, s_1 \\ \rho, \sigma_1, R \vdash e_2 \xrightarrow{\text{NESL}} v_2, \sigma_2; g_2, s_2 \end{array}}{\rho, \sigma, R \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \xrightarrow{\text{NESL}} v_2, \sigma_2; \mathbf{1} \oplus g_1 \oplus g_2, \max(s_1 + 1, s_2)} \qquad \text{(IF-TRUE)}$$

$$\frac{\begin{array}{c} \rho, \sigma, R \cup \rho(FV(e_2)) \cup \rho(FV(e_3)) \vdash e_1 \xrightarrow{\text{NESL}} \mathbf{false}, \sigma_1; g_1, s_1 \\ \rho, \sigma_1, R \vdash e_3 \xrightarrow{\text{NESL}} v_3, \sigma_3; g_3, s_3 \end{array}}{\rho, \sigma, R \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \xrightarrow{\text{NESL}} v_3, \sigma_3; \mathbf{1} \oplus g_1 \oplus g_3, \max(s_1 + 1, s_3)} \qquad \text{(IF-FALSE)}$$

$$\frac{\begin{array}{c} \rho, \sigma, R \cup \rho(FV(e')) \vdash e \xrightarrow{\text{NESL}} l, \sigma_0; g, s \qquad \sigma_0(l) = \vec{v} \qquad m = |\vec{v}| \\ \rho[x \mapsto v_j], \sigma_j, R \cup \rho(FV(e') - \{x\}) \vdash e' \xrightarrow{\text{NESL}} v'_j, \sigma_{j+1}; g_j, s'_j \qquad \forall j \in \{0, \ldots, m-1\} \end{array}}{\rho, \sigma, R \vdash \{e' : x\ \mathbf{in}\ e\} \xrightarrow{\text{NESL}} l', \sigma_m[l' \mapsto \vec{v'}]; g \oplus (\bigotimes_{j=0}^{m-1} \mathbf{1}) \oplus (\bigotimes \vec{g}), \max(s, m + \max(\vec{s'}))} \qquad \text{(EACH)}$$

$$\text{where } l' \notin dom(\sigma)$$

Figure 9.5: The profiling semantics of the NESL model using the definitions of $\delta$ and $S(\sigma, R)$ in Figures 9.6 and 9.7, respectively.

| $c$ | $v$ | $\delta(c,v,\sigma)$ | | | if/where |
|---|---|---|---|---|---|
| | | $v'$ | $\sigma'$ | $g'$ | |
| Include the definitions of Figure 5.18. | | | | | |
| | | | | $\dots$ | |
| **elt** | $l$ | $v_j$ | $\cdot$ | $\mathbf{1}$ | $\sigma(l) = \langle l',j \rangle,\ \sigma(l') = \vec{v}$ |
| **#** | $l$ | $\lvert\vec{v}\rvert$ | $\cdot$ | $\mathbf{1}$ | $\sigma(l) = \vec{v}$ |
| **index** | $i$ | $l'$ | $[l' \mapsto [0,\dots,i-1]]$ | $\bigotimes_{j=0}^{i-1} \mathbf{1}$ | $l' \notin dom(\sigma)$ |
| **dist** | $l$ | $l'$ | $[l' \mapsto \overbrace{[v,\dots,v]}^{i}]$ | $\bigotimes_{i=0}^{i} \mathbf{1}$ | $\sigma(l) = \langle v,i \rangle,\ l' \notin dom(\sigma)$ |
| **++** | $l$ | $l'$ | $[l' \mapsto \sigma(l_1) +\!\!+ \sigma(l_2)]$ | $\bigotimes_{j=0}^{m-1} \mathbf{1}$ | $\sigma(l) = \langle l_1,l_2 \rangle,\ l' \notin dom(\sigma),$<br>$m = \lvert\sigma(l_1)\rvert + \lvert\sigma(l_2)\rvert$ |
| **pack** | $l$ | $l'$ | $[l' \mapsto [v_{i_0},\dots,$<br>$v_{i_{m-1}}]]$ | $\bigotimes_{j=0}^{m-1} \mathbf{1}$ | $\sigma(l) = \vec{l},\ m = \lvert\vec{l}\rvert,\ \sigma(l_j) = \langle v_j,c_j \rangle,$<br>$\{i_0,\dots,i_{m'-1}\} = \{i \mid c_i = \mathbf{true}\},$<br>$i_0 < \dots < i_{m'-1},\ l' \notin dom(\sigma)$ |
| **put** | $l$ | $l'$ | $[l' \mapsto \vec{v}[v'_0/i_0,\dots,$<br>$v'_{m'-1}/i_{m'-1}]]$ | $(\bigotimes_{j=0}^{m-1} \mathbf{1}) \oplus$<br>$(\bigotimes_{j=0}^{m'-1} \mathbf{1})$ | $\sigma(l) = \langle l_1,l_2 \rangle,\ \sigma(l_1) = \vec{v},\ m = \lvert\vec{v}\rvert,$<br>$\sigma(l_2) = \vec{l'},\ m' = \lvert\vec{l'}\rvert,\ \sigma(l'_j) = \langle i_j,v'_j \rangle,$<br>$0 \le i_j < m,\ l' \notin dom(\sigma)$ |
| **addscan** | $l$ | $l'$ | $[l' \mapsto [\sum_{j=0}^{0} i_j,\dots,$<br>$\sum_{j=0}^{m-1} i_j]]$ | $\bigotimes_{j=0}^{m-1} \mathbf{1}$ | $\sigma(l) = \vec{i},\ m = \lvert\vec{i}\rvert,\ l' \notin dom(\sigma)$ |
| **maxscan** | $l$ | $l'$ | $[l' \mapsto [\max_{j=0}^{0} i_j,\dots,$<br>$\max_{j=0}^{m-1} i_j]]$ | $\bigotimes_{j=0}^{m-1} \mathbf{1}$ | $\sigma(l) = \vec{i},\ m = \lvert\vec{i}\rvert,\ l' \notin dom(\sigma)$ |

Figure 9.6: The $\delta$ function defining constant application for the NESL model. The substitution in the definition of **put** gives priority to the last occurrence of any duplicate indices.

$$S(R,\sigma) = \sum_{l \in L} \begin{cases} |FV(e)| + 2 & \text{if } \sigma(l) = \mathbf{cl}(\rho,x,y,e) \\ 2 & \text{if } \sigma(l) = \langle v_1,v_2 \rangle \\ |\vec{v}| & \text{if } \sigma(l) = \vec{v} \end{cases}$$

$$\text{where } L = \bigcup_{l \in R} locs(l,\sigma)$$

$$\begin{array}{lcll} locs(c, \sigma) & = & \{\} & \\ locs(l, \sigma) & = & \{l\} \cup locs(\sigma(l),\sigma) & \\ & & & \\ locs(\mathbf{cl}(\rho,x,y,e), \sigma) & = & \bigcup_{l \in L} locs(l,\sigma) & \text{where } L = \rho(FV(e) - \{x,y\}) \\ locs(\langle v_1,v_2 \rangle, \sigma) & = & locs(v_1,\sigma) \cup locs(v_2,\sigma) & \\ locs(\vec{v}, \sigma) & = & \bigcup_{i=0}^{m-1} locs(v_i,\sigma) & \text{where } m = |\vec{v}| \end{array}$$

Figure 9.7: Semantics functions used in the NESL profiling semantics for defining reachable space.

Most of these rules are similar to those of the PAL' model, differing only in the serialization of application and pairing. The only totally new semantic rule is the EACH rule. For the expression $\{e' : x \textbf{ in } e\}$ we first evaluate the binding expression $e$, which should result in a sequence $\vec{v}$. We then evaluate the body $e'$ in parallel for each element of the sequence, with each parallel branch binding $x$ to the corresponding element of the sequence.

## 9.3 Array language

To present a relatively straightforward abstract machine (Section 9.4) we first introduce the array language, ArrL. The sequence instructions of NESL translate to a similar set which allocates memory more explicitly and uses constant functions which are more primitive. ArrL includes explicit side-effects to atomically update array elements. Note that we say that NESL sequences are represented as ArrL arrays. These two data structures are semantically equivalent, but this corresponds to a common terminology distinction between data structures without and with side-effects, respectively.

The syntax of ArrL is the same as that of NESL, except that we replace the for-each expression and the constants **index**, ++, **pack**, **put**, **addscan**, and **maxscan**with the constants **store**, **new**, **fork**, **addS**, and **maxS**, as in Figure 9.8:

- Applying **store** to an array, an index, and a value writes the value into the indexed location of the array.
- Applying **new** to an integer creates and returns a new array of that length.
- Applying **fork** to an integer $i$ and a function applies the function in parallel to each of $0, \ldots, i-1$ and returns a dummy value 0. Since the **fork** function returns a dummy value, it is useful only for any side-effects of the applications.
- The new scan operations **addS** and **maxS** compute the same as their counterparts, but they perform no allocation and thus require an additional array argument in which to store their results.

Definition 9.2 defines the profiling semantics of ArrL, which is like that of NESL, except that it adds definitions for these constants.

**Definition 9.2** (ArrL **profiling semantics**) *In the* ArrL *model,* starting with the environment $\rho$, store $\sigma$, and roots $R$, the expression $e$ evaluates to value $v$ and new store $\sigma'$ with computation graph $g$ and $s$ reachable space, *or*

$$\rho, \sigma, R \vdash e \xrightarrow{\text{ArrL}} v, \sigma'; g, s,$$

*if it is derivable from the rules of Figure 9.9. The $\delta$ function for the application of constants is given in Figure 9.6. Additional functions for the space of a computation are given in Figure 9.7.*

$$\begin{array}{lclll}
c & \in & \mathit{Constants} & ::= & \ldots \mid \mathbf{elt} \mid \# \mid & \text{sequence observers} \\
 & & & & \mathbf{store} \mid & \text{sequence update} \\
 & & & & \mathbf{new} \mid & \text{sequence constructor} \\
 & & & & \mathbf{fork} \mid & \\
 & & & & \mathbf{addS} \mid \mathbf{maxS} & \text{sequence scans} \\
e & \in & \mathit{Expressions} & ::= & \ldots &
\end{array}$$

Figure 9.8: ArrL expressions. The ellipses represent the constants and expressions of Figure 4.3.

$$\frac{\begin{array}{c}\rho,\sigma,R\cup\rho(FV(e_2))\vdash e_1 \xrightarrow{\text{ArrL}} \mathbf{fork},\sigma';g_1,s_1 \qquad \rho,\sigma',R\vdash e_2 \xrightarrow{\text{ArrL}} l,\sigma_0;g_2,s_2 \\ \sigma_0(l)=\langle i,l'\rangle \qquad i\geq 0 \qquad \sigma_0(l')=\mathbf{cl}(\rho',\_,x,e_3) \\ \rho[x\mapsto j],\sigma_j,R\cup\{l'\}\vdash e_3 \xrightarrow{\text{ArrL}} v'_j,\sigma_{j+1};g'_j,s'_j \qquad \forall j\in\{0,\ldots,i-1\}\end{array}}{\rho,\sigma,R\vdash e_1\ e_2 \xrightarrow{\text{ArrL}} 0,\sigma_n;\mathbf{1}\oplus g_1\oplus g_2\oplus(\bigotimes_{j=0}^{i-1}(\mathbf{1}\oplus {g'}_j)),\max(s_1,s_2,\vec{s'})} \quad \text{(FORK)}$$

| | | $\delta(c,v,\sigma)$ | | | |
|---|---|---|---|---|---|
| $c$ | $v$ | $v'$ | $\sigma'$ | $g'$ | if/where |
| **store** | $l$ | $v'$ | $\sigma[l_1 \mapsto \vec{v}[v'/i]]$ | $\mathbf{1}$ | $\sigma(l)=\langle l_1,l_2\rangle,\ \sigma(l_1)=\vec{v},\ \sigma(l_2)=\langle i,v'\rangle$ |
| **new** | $i$ | $l$ | $\sigma[l\mapsto \overbrace{[0,\ldots,0]}^{i}]$ | $\bigotimes_{j=0}^{i-1}\mathbf{1}$ | $l\notin dom(\sigma)$ |
| **addS** | $l$ | $l''$ | $[l'\mapsto[\sum_{j=0}^{0} i_j,\ldots,\sum_{j=0}^{m-1} i_j]]$ | $\bigotimes_{j=0}^{m-1}\mathbf{1}$ | $\sigma(l)=\langle l_1,l_2\rangle,\ \sigma(l_1)=\vec{i},\ \sigma(l_2)=\vec{-},\ m=\lvert\vec{i}\rvert$ |
| **maxS** | $l$ | $l''$ | $[l'\mapsto[\max_{j=0}^{0} i_j,\ldots,\max_{j=0}^{m-1} i_j]]$ | $\bigotimes_{j=0}^{m-1}\mathbf{1}$ | $\sigma(l)=\langle l_1,l_2\rangle,\ \sigma(l_1)=\vec{i},\ \sigma(l_2)=\vec{-},\ m=\lvert\vec{i}\rvert$ |

Figure 9.9: The semantics rules for the constants added to ArrL.

Since all uses of **fork** are introduced by the translation, the value of the argument $e_2$ is guaranteed to be a non-recursive closure with the environment $\rho$. The FORK rule evaluates the function, the argument, and then the parallelized branches. Note that there is unit overhead prior to each parallel branch—this will correspond with unit work in the abstract machine to set up the evaluation of each branch.

### Equivalence of the NESL and ArrL models

The NESL and ArrL models, where the latter is restricted to using the expressions in the image of a translation, are essentially equivalent in that they can simulate each other with only a constant factor of overhead. Here we show only that the restricted ArrL model can simulate the NESL model. This simulation requires translating NESL into ArrL. The remainder of this section gives such a translation $T_{\mathrm{ArrL}}[\![]\!]$, proves that $T_{\mathrm{ArrL}}[\![]\!]$ is correct, and proves that $T_{\mathrm{ArrL}}[\![]\!]$ introduces only a constant factor of overhead. This follows the same structure as in Section 5.5.2.

Definition 9.3 defines a translation from the semantic domains of the NESL model to those of the ArrL model, and Theorem 9.1 shows that the translated evaluation derivation is correct and incurs at most a constant factor of overhead. The ArrL derivation uses an initial environment and store defining NESL's extra constants. The translation is a relation except on expressions, like $T_{\mathrm{PAL}}[\![]\!]$, as that allows it to be independent of location names.

**Definition 9.3** *Figures 9.10 and 9.11 show the translation* $T_{\mathrm{ArrL}}[\![e]\!]$ *of* NESL *expressions, values, and store-values to those of the* ArrL *model. The translations of environments and stores are defined point-wise on the values and store-values in their ranges, respectively, and the translation of root set of values is defined point-wise on the contents.*

For example, the translation of the for-each $\{e' : x \textbf{ in } e\}$ evaluates the binding expression $e$, allocates a result array, and forks $i$ threads to evaluate the body $e'$ and writes the result in the appropriate element of the result array. The motivation for separately allocating space with **new** in the translation is to account for the allocation of $k$ space with $k$ work prior to the allocation. This allows us to bound memory use (see the proof of Theorem 9.5). Note that side-effects are used only in a single-assignment manner—each location is assigned a value exactly once.

**Example 9.1** *Consider the* NESL *expression*

$$\{\textbf{add } (x,1) : x \textbf{ in index } 4\}.$$

*Its translation under* $T_{\mathrm{ArrL}}[\![]\!]$ *is*

$$\begin{array}{l} \textbf{let } x = x_{\textbf{index}}\ 4 \\ \quad\ z = \textbf{new } (\#\ x) \\ \quad\ \_ = \textbf{fork } (\#\ x, \lambda y.z[y] := (\lambda x.\textbf{add } (x,1))\ x[y]) \\ \textbf{in } z \end{array}$$

*where we use same syntactic shorthand as in the translation.*

$$\begin{array}{lcl}
T_{\mathrm{ArrL}}\llbracket \mathbf{index} \rrbracket & = & x_{\mathbf{index}} \\
T_{\mathrm{ArrL}}\llbracket \mathbf{dist} \rrbracket & = & x_{\mathbf{dist}} \\
T_{\mathrm{ArrL}}\llbracket ++ \rrbracket & = & x_{++} \\
T_{\mathrm{ArrL}}\llbracket \mathbf{pack} \rrbracket & = & x_{\mathbf{pack}} \\
T_{\mathrm{ArrL}}\llbracket \mathbf{put} \rrbracket & = & x_{\mathbf{put}} \\
T_{\mathrm{ArrL}}\llbracket \mathbf{addscan} \rrbracket & = & x_{\mathbf{addscan}} \\
T_{\mathrm{ArrL}}\llbracket \mathbf{maxscan} \rrbracket & = & x_{\mathbf{maxscan}} \\
 & & \\
T_{\mathrm{ArrL}}\llbracket \{e' : x \textbf{ in } e\} \rrbracket & = & \textbf{let } x = T_{\mathrm{ArrL}}\llbracket e \rrbracket \\
 & & \quad\;\; z = \textbf{new } (\#\ x) \\
 & & \quad\;\; \_ = \textbf{fork } (\#\ x, \lambda y.z[y] := (\lambda x.T_{\mathrm{ArrL}}\llbracket e' \rrbracket)\ x[y]) \\
 & & \textbf{in } z
\end{array}$$

$$\begin{array}{llll}
\text{where} & e_1[e_2] & \text{abbreviates} & \mathbf{elt}\ (e_1, e_2) \\
 & e_1[e_2] := e_3 & \text{abbreviates} & \mathbf{store}\ (e_1, (e_2, e_3))
\end{array}$$

Figure 9.10: Translation $T_{\mathrm{ArrL}}\llbracket\rrbracket$ from NESL expressions to those of ArrL. The translation simply translates any subexpressions, *i.e.*, is homomorphic, where not shown. The variables $x_c$ are assumed to be distinct from the free variables of the expression or closure being translated and are defined in the initial environment (Figure 9.12). For convenience, we use a multi-assignment **let** statement that executes the assignments in sequential order, which is also translated into NESL.

Values:

$$\begin{array}{ll}
T_{\mathrm{ArrL}}\llbracket c \rrbracket_{\sigma'} \llbracket c \rrbracket_{\sigma} & \text{if } c \notin \{\mathbf{index}, \mathbf{dist}, ++, \mathbf{pack}, \mathbf{put}, \mathbf{addscan}, \mathbf{maxscan}\} \\
T_{\mathrm{ArrL}}\llbracket c \rrbracket_{\sigma'} \llbracket l_c \rrbracket_{\sigma} & \text{if } c \in \{\mathbf{index}, \mathbf{dist}, ++, \mathbf{pack}, \mathbf{put}, \mathbf{addscan}, \mathbf{maxscan}\},\ l_c \in \sigma \\
T_{\mathrm{ArrL}}\llbracket l' \rrbracket_{\sigma'} \llbracket l \rrbracket_{\sigma} & \text{if } T_{\mathrm{ArrL}}\llbracket l'(\sigma') \rrbracket_{\sigma'} \llbracket l(\sigma) \rrbracket_{\sigma}
\end{array}$$

Store-Values:

$$\begin{array}{ll}
T_{\mathrm{ArrL}}\llbracket \mathbf{cl}(\rho', x, y, e) \rrbracket_{\sigma'} \llbracket \mathbf{cl}(\rho, x, y, T_{\mathrm{ArrL}}\llbracket e \rrbracket) \rrbracket_{\sigma} & \text{if } T_{\mathrm{ArrL}}\llbracket \rho' \rrbracket_{\sigma'} \llbracket \rho \rrbracket_{\sigma} \\
T_{\mathrm{ArrL}}\llbracket \langle v_1', v_2' \rangle \rrbracket_{\sigma'} \llbracket \langle v_1, v_2 \rangle \rrbracket_{\sigma} & \text{if } T_{\mathrm{ArrL}}\llbracket v_1' \rrbracket_{\sigma'} \llbracket v_1 \rrbracket_{\sigma},\ T_{\mathrm{ArrL}}\llbracket v_2' \rrbracket_{\sigma'} \llbracket v_2 \rrbracket_{\sigma} \\
T_{\mathrm{ArrL}}\llbracket \vec{v'} \rrbracket_{\sigma'} \llbracket \vec{v} \rrbracket_{\sigma} & \text{if } T_{\mathrm{ArrL}}\llbracket v_0' \rrbracket_{\sigma'} \llbracket v_0 \rrbracket_{\sigma}, \ldots, T_{\mathrm{ArrL}}\llbracket v_{k-1}' \rrbracket_{\sigma'} \llbracket v_{k-1} \rrbracket_{\sigma} \\
 & \quad \text{where } k = |\vec{v'}| = |\vec{v}|
\end{array}$$

Figure 9.11: Translation $T_{\mathrm{ArrL}}\llbracket\rrbracket$ from NESL values and store-values to those of ArrL, using the translation on expressions.

| $\rho_I$ | | |
|---|---|---|
| $x_c$ | $\mapsto$ | $l_c$, for each $c \in \{\mathbf{index}, \mathbf{dist}, \mathbf{++}, \mathbf{pack}, \mathbf{put}, \mathbf{addscan}, \mathbf{maxscan}\}$ |

$\sigma_I$

$$
\begin{array}{rcl}
l_c & \mapsto & \mathbf{cl}(\cdot,\_,x,e_c),\ \text{for each } c \in \{\mathbf{index}, \mathbf{dist}, \mathbf{++}, \mathbf{pack}, \mathbf{put}, \mathbf{addscan}, \mathbf{maxscan}\} \\
\\
e_{\mathbf{index}} & = & \mathbf{let}\ z = \mathbf{new}\ x \\
 & & \quad \_ = \mathbf{fork}\ (x,\lambda y.z[y] := y) \\
 & & \mathbf{in}\ z \\
e_{\mathbf{dist}} & = & \mathbf{let}\ x_1 = \mathbf{fst}\ x \\
 & & \quad x_2 = \mathbf{snd}\ x \\
 & & \quad z = \mathbf{new}\ x_2 \\
 & & \quad \_ = \mathbf{fork}\ (x_2,\lambda y.z[y] := x_1) \\
 & & \mathbf{in}\ z \\
e_{\mathbf{++}} & = & \mathbf{let}\ x_1 = \mathbf{fst}\ x \\
 & & \quad x_2 = \mathbf{snd}\ x \\
 & & \quad z = \mathbf{new}\ (\mathbf{add}\ (\#\ x_1,\#\ x_2)) \\
 & & \quad \_ = \mathbf{fork}\ (x_2,\lambda y.z[y] := (\mathbf{if}\ \mathbf{lt}\ (y,\#\ x_1)\ \mathbf{then}\ x_1[y]\ \mathbf{else}\ x_2[\mathbf{sub}\ (y,\#\ x_1)])) \\
 & & \mathbf{in}\ z \\
e_{\mathbf{pack}} & = & \mathbf{let}\ x_1 = \mathbf{fst}\ x \\
 & & \quad x_2 = \mathbf{snd}\ x \\
 & & \quad x' = T_{\text{ArrL}}[\![\mathbf{addscan}]\!]\ T_{\text{ArrL}}[\![\{\mathbf{if}\ x'\ \mathbf{then}\ 1\ \mathbf{else}\ 0 : x'\ \mathbf{in}\ x_2\}]\!] \\
 & & \quad z = \mathbf{new}\ x'[\mathbf{sub}\ (\#\ x)\ 1] \\
 & & \quad \_ = \mathbf{fork}\ (\#\ x,\lambda y.\mathbf{if}\ \mathbf{snd}\ (x_1[y])\ \mathbf{then}\ z[x'[y]] := \mathbf{fst}\ x_1[y]\ \mathbf{else}\ 0) \\
 & & \mathbf{in}\ z \\
e_{\mathbf{put}} & = & \mathbf{let}\ x_1 = \mathbf{fst}\ x \\
 & & \quad x_2 = \mathbf{snd}\ x \\
 & & \quad z = \mathbf{new}\ (\#\ x_1) \\
 & & \quad \_ = \mathbf{fork}\ (\#\ x_1,\lambda y.z[y] := x_1[y]) \qquad \text{(copy destination array)} \\
 & & \quad \_ = \mathbf{fork}\ (\#\ x_2,\lambda y.z[\mathbf{fst}\ x_2[y]] := \mathbf{snd}\ x_2[y]) \\
 & & \mathbf{in}\ z \\
e_{\mathbf{addscan}} & = & \mathbf{addS}\ (x,\mathbf{new}\ (\#\ x)) \\
e_{\mathbf{maxscan}} & = & \mathbf{maxS}\ (x,\mathbf{new}\ (\#\ x))
\end{array}
$$

$$
\begin{array}{lll}
\text{where}\ e_1[e_2] & \text{abbreviates} & \mathbf{elt}\ (e_1,e_2) \\
\qquad e_1[e_2] := e_3 & \text{abbreviates} & \mathbf{store}\ (e_1,(e_2,e_3))
\end{array}
$$

Figure 9.12: Initial ArrL environment $\rho_I$ and store $\sigma_I$ when translating from NESL with $T_{\text{ArrL}}[\![]\!]$. For convenience, we use a multi-assignment **let** statement that executes the assignments in sequential order, which is also translated into NESL.

Theorem 9.1 now shows that the PAL model can simulate the PAL' model with only a constant factor of overhead. To prove this, Lemma 9.1 shows that the simulation holds for all contexts. These are very similar to Theorem 5.1 and Lemma 5.1, so we do not provide as many details in these proofs.

**Lemma 9.1 (Equivalence of** NESL **and** ArrL**)** *If e evaluates in the* NESL *model:*

$$\rho', \sigma', R' \vdash e \overset{\text{NESL}}{\longrightarrow} v', \sigma' \cup \sigma'_{new}; g', s',$$

*then for any context of $\rho$, $\sigma$, and $R$ for the corresponding* ArrL *derivation such that*

- *its initial context is the translation of that of the* NESL *derivation: $T_{\text{ArrL}}\llbracket\rho'\rrbracket_{\sigma'}\llbracket\rho\rrbracket_{\sigma}$, $T_{\text{ArrL}}\llbracket\sigma'\rrbracket\llbracket\sigma\rrbracket$, $T_{\text{ArrL}}\llbracket R'\rrbracket_{\sigma'}\llbracket R\rrbracket_{\sigma}$, $S(R,\sigma) \leq k \cdot S(R',\sigma')$, and*
- *it uses the initial environment and initial store defined in Figure 9.12: $\rho_I \cup \rho$ and $\sigma_I \cup \sigma$,*

*then e's translation evaluates in the* ArrL *model:*

$$\rho_I \cup \rho, \sigma_I \cup \sigma, R \vdash T_{\text{ArrL}}\llbracket e\rrbracket \overset{\text{ArrL}}{\longrightarrow} v, \sigma \cup \sigma_{new}; g, s$$

*such that*

- *it results in the translated value: $T_{\text{ArrL}}\llbracket v'\rrbracket_{\sigma' \cup \sigma'_{new}}\llbracket v\rrbracket_{\sigma \cup \sigma_{new}}$, and*
- *its costs are at most a constant factor more than those of the* NESL *evaluation: $W(g') \leq k \cdot W(g)$, $D(g') \leq k \cdot D(g)$, and $s' \leq k \cdot s$, for some constant $k$.*

*Proof Outline:* We prove this by induction on the structure of the NESL evaluation derivation. We assume that the NESL derivation holds and prove the ArrL derivation and side conditions hold, using a case analysis on the last rule used in the NESL derivation. The definition of the translation $T_{\text{ArrL}}\llbracket\rrbracket$ on environments and stores make most cases entirely straightforward.

The second condition on the ArrL context holds inductively since, by definition, the domains of the initial environment and store are distinct from any other variables or locations. In most cases, the first conclusion holds by simple observation of the definition of the translation. The second conclusion holds by showing the translation introduces only a constant factor larger computation graph and a constant factor of extra closures.

**case CONST, $e = c$:** If $T_{\text{ArrL}}\llbracket c\rrbracket = c$, then the conclusion holds since the NESL constant rule corresponds to the ArrL constant rule.

Otherwise, $T_{\text{ArrL}}\llbracket c\rrbracket = l_c$, and the conclusion follows from the definition of the bindings of the initial environment and store. It holds since the NESL constant rule corresponds to an ArrL variable lookup, so $g' = g$ and $s < k \cdot s'$ for some constant $k$ determined by the size of the closures in $\sigma_I$.

**case LAM,** $e = \lambda x.e'$**:** The conclusion holds since the NESL abstraction rule corresponds to the ArrL abstraction rule.

**case VAR,** $e = x$**:** This follows from the definition of the translation of an environment, since by definition $T_{\text{ArrL}}[\![\rho'(x)]\!]_{\sigma'}[\![\rho(x)]\!]_{\sigma}$ holds if $T_{\text{ArrL}}[\![\rho']\!]_{\sigma'}[\![\rho]\!]_{\sigma}$ holds. The conclusion follows since the NESL variable lookup corresponds exactly to an ArrL variable lookup, of a value that is at most a constant factor larger than the NESL value.

**case APPC,** $e = e_1\ e_2$**:** By assumption, the function $e_1$ evaluates to a constant $c$. By induction, the conclusion holds for both subexpressions, and in particular, we obtain the corresponding graphs $g_1$ and $g_2$. The exact structure of the ArrL evaluation depends upon the constant $c$. If $T_{\text{ArrL}}[\![c]\!] = c$, this last step of the NESL and ArrL derivations correspond exactly. Otherwise, $T_{\text{ArrL}}[\![c]\!]$ is an abstraction, and in each case the ArrL application involves a constant amount of overhead.

**cases APP, PAIR, LETREC, IF-TRUE, and IF-FALSE:** Since the translation is homomorphic on applications, pairs, recursive definitions, and conditional branches,

- the conclusion holds inductively on each subexpression, and
- this last step of the NESL and ArrL derivations correspond exactly.

**case EACH,** $e = \{e'' : x \textbf{ in } e'\}$**:** By assumption, $e''$ evaluates to a location that contains an array. By induction, the conclusion holds for $e'$ and for each of the $m$ evaluations of $e''$, and in particular, we obtain the graphs $g'$ and $g''_0, \ldots, g''_{m-1}$. The translation introduces a constant factor of overhead, as illustrated by Figure 9.13. This includes several applications, including those of **new** and **fork**.

□

**Theorem 9.1 (Equivalence of** NESL **and** ArrL**)** *If e evaluates in the* NESL *model:*

$$\cdot, \cdot, \{\} \vdash e' \overset{\text{NESL}}{\longrightarrow} v', \sigma'; g', s',$$

*and the corresponding* ArrL *derivation uses the appropriately translated expression:*

$$e = (\lambda x_{\textbf{index}} \ldots \lambda x_{\textbf{maxscan}}.T_{\text{ArrL}}[\![e']\!])\ (\lambda x.e_{\textbf{index}}) \ldots (\lambda x.e_{\textbf{maxscan}})$$

*using the subexpressions defined in Figure 9.12, then the translation of $e'$ evaluates in the* ArrL *model:*

$$\cdot, \cdot, \{\} \vdash e \overset{\text{ArrL}}{\longrightarrow} v, \sigma; g, s$$

*such that*

- *it results in the translated value:* $T_{\text{ArrL}}[\![v']\!]_{\sigma'}[\![v]\!]_{\sigma}$, *and*

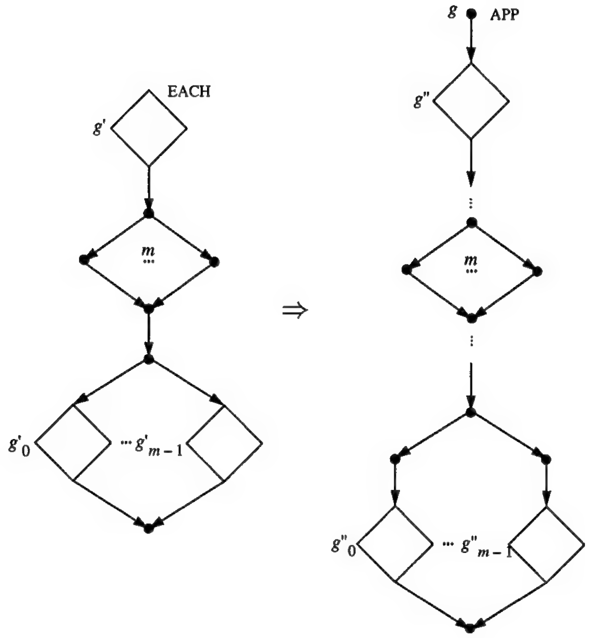


Figure 9.13: The ArrL computation graph $g$ corresponding to that produced by the NESL EACH rule.

- *its costs are at most a constant factor more than those of the* NESL *model:* $W(g') \leq k \cdot W(g)$, $D(g') \leq k \cdot D(g)$, *and* $s' \leq k \cdot s$, *for some constant* $k$.

*Proof:* This follows from Lemma 9.1. The initial applications in $e$ set up the initial environment and store for that lemma. □

We are only interested in ArrL expressions obtained via this translation. This places two constraints of interest on expressions:

- Since each translation rule introduces side-effects only on local variables introduced by the translation, not including the function parameters, these side-effects cannot interfere with each other. Again note that the side-effects are single-assignment. Thus the translation does not introduce nondeterminism. Therefore the constraint ensures that the expressions are deterministic, despite the presence of side-effects.

- It ensures that the second argument of **fork** is an abstraction. Thus its value uses the current defining environment, simplifying the FORK case of the proof of Lemma 9.2.

## 9.4 Intermediate model

The P-CEK$^q_{\text{ArrL}}$ abstract machine is similar to the P-CEK$^q_{\text{PAL}}$ machine. It executes a series of steps, each transforming a group of active states and a store into a new group of active states and new store for the next step. Figure 9.14 illustrates this process. It starts with with one active state for the entire computation, selects up to $q$ active states per step, and ends with one active state with the final value. Bounds on execution costs hold for the same reason as before: the machine executes a $q$-DFT of the computation graph returned by the profiling semantics, and its reachable space is bounded by the serial space returned by the profiling semantics.

For brevity, we omit **maxS**, pairing, conditionals, and recursive bindings from our definitions and proofs. The primitive **maxS** would be treated similarly to **addS**. Pairing, conditionals, and recursive bindings are all serialized in this model, and thus are relatively uninteresting. We present further details of this omission as relevant.

Like the P-CEK$^q_{\text{PAL}}$, the machine uses the expressions @ $v_1$ $v_2$ and **done** $v$ to represent the function body and the end of the entire computation, respectively. The additional expression $\Sigma\ l_1\ l_2\ l'\ i$ represents part of a **addS** computation: the addition of the $i$th element of the array at $l_1$ to the result array at $l_2$, where the current total is stored at $l'$. A similar expression would be included for the omitted **maxS**.

The primary difference between the P-CEK$^q_{\text{ArrL}}$ and the previous abstract machines is that it can create an unbounded number of threads in a single step. But allocating the space for an unbounded number of active states all at once would break our space bounds. So we must ensure that we allocate space for no more than $O(q)$ active states on each step. The constructs which can allocate unbounded number of computations are **addS** and **fork** (and

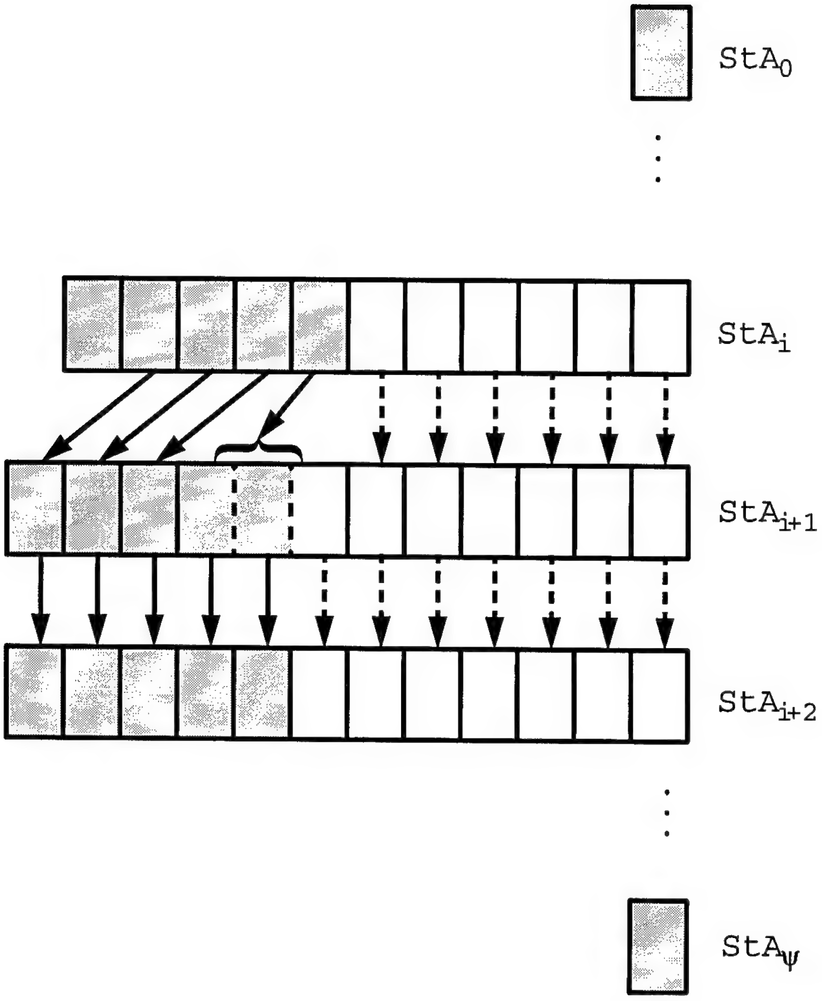


Figure 9.14: Illustration of P-CEK$^q_{\mathrm{ArrL}}$ active states during an evaluation. It starts with one active states representing the entire program and ends with one active states representing the result value. The states are kept in a stack. At most $q$ states are selected each step. Here, $q = 5$, and these selected states are shaded. These can create zero or one new state, including the possibility of a stub representing multiple states (solid arrows). Unselected states are still active in the next step (dashed arrows).

the omitted **maxS**). Rather than immediately creating an active state per created thread, we create a single *stub* state. When selecting active states for a step, we expand the stubs to the relevant states; if the step does not need all of a stub's representative states, it places a new stub on the active states stack to represent the remaining states. The stub $@\mathsf{stub}\langle l\ i\ k\ \kappa\rangle$ represents states of a **fork** call, where $l$ is a pointer to the function value (closure) to be called on each of the remaining numbers $i, \ldots, k-1$, and $\kappa$ is the continuation to be used after these operations. The stub $\Sigma\mathsf{stub}\langle l_1\ l_2\ l'\ i\ k\ \kappa\rangle$ represents states of an **addS** call, where $l_1$ is a pointer to the source array to be added, $l_2$ is a pointer to the result array, $l'$ is a pointer to the running summation total, and $i, \ldots, k-1$ are the remaining indices of data to sum.

Each step consists of selecting these states, and then one substep for computation and one substep for communication and synchronization.

This machine uses four forms of continuation. The empty continuation $\bullet$ represents the end the entire computation. The continuation $\mathsf{arg}\langle e\ \rho\ \kappa\rangle$ represents the need to evaluate the argument $e$ of an application after evaluating the corresponding function. The continuation $\mathsf{fun}\langle v\ \kappa\rangle$ represents the need to apply the function value $v$ to what is currently being evaluated. And, finally, the continuation $\mathsf{end}\langle l\ \kappa\rangle$ represents the need to synchronize threads (originally spawned by **fork**) on the counter in $l$. Additional continuation forms are needed to serialize the omitted pairing, conditional, and recursive binding expressions.

We implement the semantics' nested sequences as nested sequences in the abstract machine, unlike in the current implementation of NESL [14]. During compilation, NESL *flattens* nested sequences and any code building or using nested sequences. As a result, NESL's implementation (*i.e.*, VCODE) uses only larger, one-dimensional sequences. This has the advantage of increasing the granularity for parallelism, which is very important in practice. But it has the disadvantages of increasing sequence sizes, and thus the cost of applying most constant functions, and of not allowing subsequences to be shared.

Given the domains just described, as defined in Figure 9.15, Definitions 9.4 and 9.5 then define a P-CEK$^q_{\mathrm{ArrL}}$ step and the abstract machine. Remember that this omits **maxS**, pairing, conditionals, and recursive bindings.

**Definition 9.4 (P-CEK$^q_{\mathrm{ArrL}}$ step)** *A* step $i$ of the P-CEK$^q_{\mathrm{ArrL}}$ machine*, written*

$$StA_i, \sigma_i \overset{\mathrm{ArrL},q}{\hookrightarrow} StA_{i+1}, \sigma_{i+1}; Q_i, s_i,$$

*is defined in Figure 9.16. It starts with a stack of active states $StA_i$ and a store $\sigma_i$ and produces a new stack and store for the next step. This step requires processes $Q_i$ (non-stub) states and uses $s_i$ maximum reachable space.*

Note that we do not count stubs in the number of states processed by the machine. We are only interested in the count of non-stub states because they will correspond to the nodes of the computation graph. The stubs are only important for the efficient storage of the active state stack.

| | | | | | |
|---|---|---|---|---|---|
| $e$ | $\in$ | *Expressions* | $::=$ | $\ldots \mid @\ v_1\ v_2 \mid$ | application |
| | | | | $\mathbf{done}\ v \mid$ | final result |
| | | | | $\Sigma\ l_1\ l_2\ l'\ i$ | addscan element |
| $C$ | $\in$ | *Controls* | $::=$ | $e$ | |
| $\rho$ | $\in$ | *Environments* | $=$ | $\mathit{Variables} \xrightarrow{fin} \mathit{Values}$ | |
| $\kappa$ | $\in$ | *Continuations* | $::=$ | $\bullet \mid$ | program finishing |
| | | | | $\mathsf{fun}\langle v\ \kappa\rangle \mid$ | function finishing |
| | | | | $\mathsf{arg}\langle e\ \rho\ \kappa\rangle \mid$ | argument finishing |
| | | | | $\mathsf{end}\langle l\ \kappa\rangle$ | for-each branch finishing |
| $st$ | $\in$ | *States* | $::=$ | $(e, \rho, \kappa) \mid$ | |
| | | | | $@\mathsf{stub}\langle l\ i\ k\ \kappa\rangle \mid$ | for-each stub |
| | | | | $\Sigma\mathsf{stub}\langle l_1\ l_2\ l'\ i\ k\ \kappa\rangle$ | addscan stub |
| $St$ | $\in$ | *StateArrays* | $::=$ | $\vec{st}$ | |
| $I$ | $\in$ | *IntermediateStates* | $::=$ | $St \mid$ | new states |
| | | | | $\mathsf{Fin}\langle l\ \kappa\rangle \mid$ | finishing state |
| | | | | $\Sigma\langle l_1, l_2, l', i, \kappa\rangle$ | addscan |
| $\sigma$ | $\in$ | *Stores* | $=$ | $\mathit{Locations} \xrightarrow{fin} (\mathit{StoreValues}+$ $\mathit{Integers})$ | |

Figure 9.15: P-CEK$^q_{\mathrm{ArrL}}$ domains. The ellipsis represents the expressions of Figure 4.3.

**Definition 9.5 (P-CEK$^q_{\mathrm{ArrL}}$ evaluation)** *In the P-CEK$^q_{\mathrm{ArrL}}$ machine, the* evaluation of expression $e$ to value $v$, starting in the environment $\rho$ and store $\sigma_0$, ends with store $\sigma_\psi$, processing $Q$ states in $\psi$ parallel steps using $s$ maximum reachable space, *or*

$$\rho, \sigma_0 \vdash e \overset{\mathrm{ArrL},q}{\Longrightarrow} v, \sigma_\psi; Q, \psi, s.$$

*For each of these $i \in \{0, \ldots, \psi - 1\}$ steps,*

$$StA_i, \sigma_i \overset{\mathrm{ArrL},q}{\hookrightarrow} StA_{i+1}, \sigma_{i+1}; Q_i, s_i,$$

*such that*

- *the machine starts with one active state for the whole program: $StA_0 = [(e, \rho, \bullet)]$, $\sigma_0 = \cdot$;*
- *the machine ends with one active state with the result value: $StA_\psi = [(\mathbf{done}\ v, \cdot, \bullet)]$; and*
- *the total number of states processed and maximum reachable space are $Q = \sum_{i=0}^{\psi-1} Q_i$ and $s = \max_{i=0}^{\psi-1} s_i$.*

As in the other models, each step begins by selecting at most $q$ states. But here we must allow the stubs to expand to the ordered set of states they represent. If the step does not need all of a stub's states, the machine puts a stub representing the extra states back on the active states stack. The stub expansion is defined by Figure 9.17. When selected, a for-each stub sets up the application of the closure at location $l$ to the dummy value in the stub[2], and an addscan stub sets up the addition of the next element. While defined sequentially, the machine parallelizes the selection of multiple elements.

This abstract machine uses only two substeps per step: one for computation and one for communication and synchronization, as illustrated in Figure 9.18. The computation step parallelizes the $\overset{\mathrm{ArrL}}{\hookrightarrow}_{comp}$ transition. This transition corresponds to that of the PAL model, with two main differences:

- the **fork** and **addS** primitive function calls create stub states, and
- the expression $\Sigma\ l_1\ l_2\ l'\ i$ leads to the addition of the $i$th element of an **addS** application.

The communication substep parallelizes the $\overset{\mathrm{ArrL}}{\hookrightarrow}_{sync}$ transition. This transition synchronizes the results of the **fork** and scan applications. Each branch of a **fork** ends with a $\mathsf{Fin}\langle l\ \kappa\rangle$ intermediate state. These branches may complete any arbitrary order; $l$ is a synchronization counter that indicates how many branches have yet to finish. The transition decrements the

[2] We know it is a closure by definition of the translation.

$$
\begin{array}{llllllll}
\multicolumn{2}{c}{st} & & & I & & \text{if/where} \\
(c, & -,\ \kappa) & - & \overset{\text{ArrL}}{\hookrightarrow}_{comp} & throw(c,\kappa) & \cdot & \\
(x, & \rho,\ \kappa) & - & \overset{\text{ArrL}}{\hookrightarrow}_{comp} & throw(\rho(x),\kappa) & \cdot & \\
(\lambda x.e, & \rho,\ \kappa) & \sigma & \overset{\text{ArrL}}{\hookrightarrow}_{comp} & throw(l,\kappa) & [l \mapsto \mathbf{cl}(\rho',\_,x,e)] & \rho' = restr(\rho,\lambda x.e), \\
 & & & & & & l \notin dom(\sigma) \\
(e_1\ e_2, & \rho,\ \kappa) & \sigma & \overset{\text{ArrL}}{\hookrightarrow}_{comp} & [(e_1,\rho,\mathsf{arg}\langle e_2\ \rho'\ \kappa\rangle)] & \cdot & \rho' = restr(\rho,e_2) \\
(@\ l\ v, & \cdot,\ \kappa) & \sigma & \overset{\text{ArrL}}{\hookrightarrow}_{comp} & [(e,\rho'[x\mapsto l][y\mapsto v],\kappa)] & \cdot & l(\sigma) = \mathbf{cl}(\rho',x,y,e) \\
(@\ \mathbf{fork}\ l, & \cdot,\ \kappa) & \sigma & \overset{\text{ArrL}}{\hookrightarrow}_{comp} & [@\mathsf{stub}\langle l'\ 0\ i\ \mathsf{end}\langle l''\ \kappa\rangle\rangle] & [l''\mapsto i-1] & \sigma(l) = \langle i,l'\rangle,\ i\geq 0, \\
 & & & & & & l''\notin dom(\sigma) \\
(@\ \mathbf{addS}\ l, & \cdot,\ \kappa) & \sigma & \overset{\text{ArrL}}{\hookrightarrow}_{comp} & [\Sigma\mathsf{stub}\langle l_1\ l_2\ l'\ 0\ m\ \kappa\rangle] & [l'\mapsto 0] & \sigma(l) = \langle l_1,l_2\rangle, \\
 & & & & & & m = |\sigma(l_1)|, \\
 & & & & & & l'\notin dom(\sigma) \\
(@\ c\ v, & \cdot,\ \kappa) & \sigma & \overset{\text{ArrL}}{\hookrightarrow}_{comp} & throw(v',\kappa) & \sigma' & \delta(c,v,\sigma) = v',\sigma';- \\
(\Sigma\ l_1\ l_2\ l'\ i, & \cdot,\ \kappa) & \sigma & \overset{\text{ArrL}}{\hookrightarrow}_{comp} & \Sigma\langle l_1,l_2,l',i,\kappa\rangle & \cdot &
\end{array}
$$

$$
\begin{array}{llcl}
\text{where} & throw(v,\ \bullet) & = & [(\mathbf{done}\ v,\cdot,\bullet)] \\
 & throw(v,\ \mathsf{arg}\langle e\ \rho\ \kappa\rangle) & = & [(e,\rho,\mathsf{fun}\langle v\ \kappa\rangle)] \\
 & throw(v_2,\ \mathsf{fun}\langle v_1\ \kappa\rangle) & = & [(@\ v_1\ v_2,\cdot,\kappa)] \\
 & throw(v,\ \mathsf{end}\langle l\ \kappa\rangle) & = & \mathsf{Fin}\langle l\ \kappa\rangle \\
 & & & \\
 & restr(\ \rho,\ \ e) & = & \text{the environment } \rho \text{ restricted to the free variables in } e
\end{array}
$$

$$
\begin{array}{lllllll}
I & & & st & & & \\
\mathsf{Fin}\langle l\ \kappa\rangle & \sigma & \overset{\text{ArrL}}{\hookrightarrow}_{sync} & [(0,\cdot,\kappa)] & \sigma & \text{if } \sigma(l)=0 & \text{last thread alive of fork} \\
 & & & [] & \sigma[l\mapsto\sigma(l)-1] & \text{if } \sigma(l)\neq 0 & \\
\Sigma\langle l_1,l_2,l',i,\kappa\rangle & \sigma & \overset{\text{ArrL}}{\hookrightarrow}_{sync} & [(x,[x\mapsto l_2],\kappa)] & \sigma' & \text{if } i=|\vec{i}|-1 & \text{last element to sum} \\
 & & & [] & \sigma' & \text{if } i\neq|\vec{i}|-1 & \\
 & & & \multicolumn{4}{l}{\text{where } \sigma(l_1)=\vec{i},\ \sigma(l')=j,\ \sigma'=\sigma[l_2\mapsto\sigma(l_2)[j/i]][l'\mapsto j+i_i]} \\
st & \sigma & \overset{\text{ArrL}}{\hookrightarrow}_{sync} & st & \sigma & &
\end{array}
$$

$$
StA,\sigma \overset{\text{ArrL},q}{\hookrightarrow} (+\!\!\!+\vec{St}){+\!\!+}StA',\sigma'_{q'};q',S_r(StA,\sigma)
$$

$$
\begin{array}{lllllll}
\text{if} & \vec{st}, & StA' & = & select(q,StA) & & \text{select at most } q \text{ active states} \\
 & q' & & = & |\vec{st}| & & \\
 & st_i, & \sigma & \overset{\text{ArrL}}{\hookrightarrow}_{comp} & I_i, & \sigma_i & \text{for each } i\in\{0,\ldots,q'-1\} \\
 & I_i, & \sigma'_i & \overset{\text{ArrL}}{\hookrightarrow}_{sync} & St_i, & \sigma'_{i+1} & \text{for each } i\in\{0,\ldots,q'-1\} \qquad \sigma'_0=\sigma\cup(\bigcup\vec{\sigma})
\end{array}
$$

Figure 9.16: Definition of the P-CEK$^q_{\text{ArrL}}$ abstract machine step, omitting some transitions. Assume all new locations of the computation step are chosen or renamed to be distinct.

| $q$ | $StA$ | $select(q, StA)$ |
|---|---|---|
| 0 | – | $[], []$ |
| 1 | $[\text{@stub}\langle l\ k\ k\ \kappa\rangle]{+\!+}St$ | $select(1, St)$ |
| 1 | $[\text{@stub}\langle l\ i\ k\ -\rangle]{+\!+}St$ | $[(@\ l\ i, \cdot, \kappa)], [\text{@stub}\langle l\ (i+1)\ k\ \kappa\rangle]{+\!+}St$<br>if $i < k$ |
| 1 | $[\Sigma\text{stub}\langle -\ -\ -\ k\ k\ -\rangle]{+\!+}St$ | $select(1, St)$ |
| 1 | $[\Sigma\text{stub}\langle l_1\ l_2\ l'\ i\ k\ \kappa\rangle]{+\!+}St$ | $[(\Sigma\ l_1\ l_2\ l'\ i, \cdot, \kappa)], [\Sigma\text{stub}\langle l_1\ l_2\ l'\ (i+1)\ k\ \kappa\rangle]{+\!+}St$<br>if $i < k$ |
| 1 | $[st]{+\!+}St$ | $[st], St$<br>if $st$ not a stub |
| $i$ | $St$ | $St_1{+\!+}St_2, St'_2$<br>if $i > 1$, $select(1, St) = St_1, St'_1$,<br>$select(i-1, St'_1) = St_2, St'_2$ |

Figure 9.17: Selecting P-CEK$^q_{\text{ArrL}}$ active states in the presence of stub states.

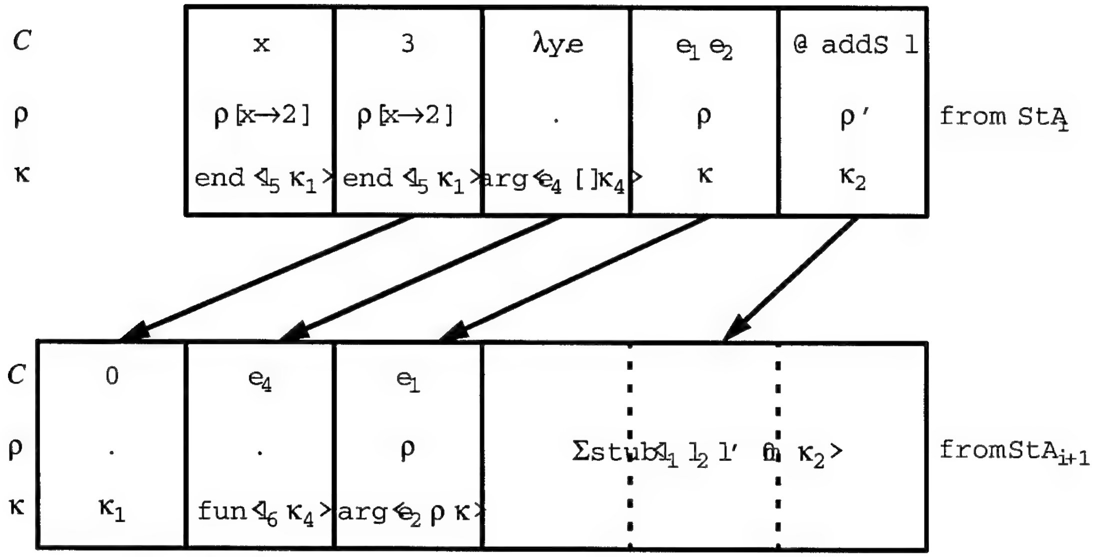
counter that was initially set to the number of branches of the **fork** call. The counter reaches zero on the last branch to finish, which generates a state to use the continuation. This state corresponds to the for-each's sink node in Figure 9.2. The machine generates an intermediate state $\Sigma\langle l_1, l_2, l', i, \kappa\rangle$ for each $i$th array element of the scan operation. The transition adds the $i$th element of the array in $l_1$ to the running total in $l'$ and stores the current total in the $i$th element of the array in $l_2$. When the machine reaches the last element, it creates a state to pass the result to the continuation. The machine does not need a synchronization location to determine which is the last element of the scan because its adds the elements in sequential order. While described serially, this substep parallelizes over all of the intermediate states generated by the first substep.

**Example 9.2** *As an example of the execution of the P-CEK$^q_{\text{ArrL}}$, we describe the active states at the beginning of each step of evaluation the following expression:*

```
let z = new 4
    _ = fork (4,λy.store (z,(y,add (y,1))))
in z
```

*We do not show the active states for each step because of the size of the example. Note that this expression is an optimization of that from Example 9.1.*

*Figure 9.19 shows the computation graph of the corresponding profiling semantics evaluation for comparison, using the appropriate states' expressions as node labels. For any value of $q$, the machine processes 85 states. For $q = 2$, this requires 51 steps, while for $q \geq 4$, it requires 34 steps. Each of these executions is a $q$-DFT of the graph, and thus for $q \geq 4$, it is also a level-order traversal. Also, in this example which has no conditionals, each parallel branch executes the same instructions.*



$$
\begin{array}{llll}
\text{where} & \sigma_i(l) & = & \langle l_1,l_2\rangle \\
 & m & = & |\sigma_i(l_1)| \\
 & l', l_6 & \notin & dom(\sigma_i) \\
 & \sigma_{i+1} & = & \sigma_i[l_6 \mapsto \mathbf{cl}(\cdot,\_,y,e)][l' \mapsto 0]
\end{array}
$$

Figure 9.18: Illustration of a P-CEK$^q_{\text{ArrL}}$ step. States with constants, variables, and abstractions create zero or one new state depending on their continuation and whether they are the last one of a fork or scan operation. States with applications create one new state to evaluate their functions first.

To formally define the reachable space during evaluation, we again consider its two components: the control space, for the control information of the active states, and the store space, for the elements in the store. We include the space for the synchronization locations in the control space (*e.g.*, $L_\kappa(\mathsf{fun}\langle l\ \kappa\rangle)$ does not add $l$ to the labels) even though they are are kept in the store so that the locations in in the profiling semantics correspond exactly to those in the P-CEK$^1_{\text{ArrL}}$ machine.

**Definition 9.6 (Reachable space of P-CEK$^q_{\text{ArrL}}$ step)** *The* reachable space of a step $i$ of the P-CEK$^q_{\text{ArrL}}$ machine, *written $S_r(StA_i, \sigma_i)$, is the sum of*

- *the active states space $S_A(StA_i)$ for the active states, including any environments and continuations: the sum of, for each state in $StA_i$,*
    - $1 + |dom(\rho)| + |\kappa|$ *for a state $(e, \rho, \kappa)$, where $|\kappa|$ is the length of the continuation stack $\kappa$; or*
    - $1 + |\kappa|$ *for a stub* $@\mathsf{stub}\langle - - - \kappa\rangle$ *or* $\Sigma\mathsf{stub}\langle - - - - - \kappa\rangle$

  *and*

- *the store space $S_\sigma(StA_i, \sigma_i)$ for program variables and all temporary values: equals the space in the store reachable from the active states used as roots, $S(L(StA_i), \sigma_i)$, where $S(-,-)$ and $L(-)$ are defined in Figures 5.14 and 9.20, respectively.*

### 9.4.1 Equivalence of language and intermediate models

In this section we relate the P-CEK$^q_{\text{ArrL}}$ to the ArrL profiling semantics. In addition to proving its extensional correctness, we also prove bounds on the time and space taken by the P-CEK$^q_{\text{ArrL}}$ machine as a function of the work, depth, and space given by the profiling semantics. As before, we prove both serial and parallel equivalence, including that the abstract machine executes a $q$-DFT of the profiling semantics' computation graph.

#### Serial equivalence

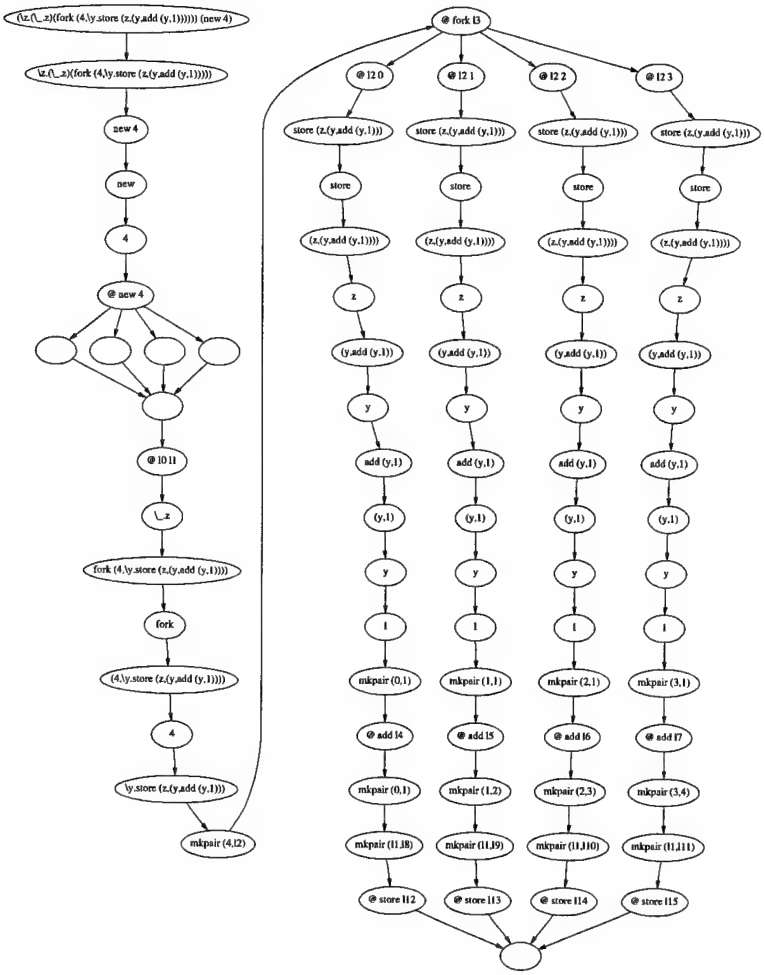
**Theorem 9.2** (ArrL **serial evaluation)** *If $e$ evaluates in the profiling semantics:*

$$\cdot, \cdot, \{\} \vdash e \overset{\text{ArrL}}{\longrightarrow} v, \sigma'; g, s,$$

*then it also evaluates in the serial abstract machine:*

$$\cdot, \cdot \vdash e \overset{\text{ArrL},1}{\Longrightarrow} v, \sigma_\psi; Q, \psi, s',$$

*such that $s' \leq k \cdot s$, for some constant $k$.*



where $l_0$ contains $\mathbf{cl}(\cdot,\_,z,-)$ $l_8$ contains $\langle 0{,}1\rangle$
$l_1$ contains $[0,0,0,0]$ $l_9$ contains $\langle 1{,}2\rangle$
$l_2$ contains $\mathbf{cl}(-,\_,\_,z)$ $l_{10}$ contains $\langle 2{,}3\rangle$
$l_3$ contains $\langle 4,l_2\rangle$ $l_{11}$ contains $\langle 3{,}4\rangle$
$l_4$ contains $\langle 0{,}1\rangle$ $l_{12}$ contains $\langle l_1,l_8\rangle$
$l_5$ contains $\langle 1{,}1\rangle$ $l_{13}$ contains $\langle l_1,l_9\rangle$
$l_6$ contains $\langle 2{,}1\rangle$ $l_{14}$ contains $\langle l_1,l_{10}\rangle$
$l_7$ contains $\langle 3{,}1\rangle$ $l_{15}$ contains $\langle l_1,l_{11}\rangle$
and **mkpair** $(v_1, v_2)$ is the omitted machine expression that creates a pair

Figure 9.19: ArrL computation graph for Example 9.2.

$$L(StA) \quad = \quad \bigcup_{st \in StA} \begin{cases} L_\rho(\rho) \cup L_\kappa(\kappa) & \text{if } st = (-, \rho, \kappa) \\ \{l\} \cup L_\kappa(\kappa) & \text{if } st = \mathsf{@stub}\langle l \; - \; - \; \kappa\rangle \\ \{l_1, l_2, l'\} \cup L_\kappa(\kappa) & \text{if } st = \Sigma\mathsf{stub}\langle l_1 \; l_2 \; l' \; - \; - \; \kappa\rangle \end{cases}$$

$$L_\rho(\rho) \quad = \quad rng(\rho)$$

$$\begin{aligned} L_\kappa(\bullet) \quad &= \quad \{\} \\ L_\kappa(\mathsf{fun}\langle v \; \kappa\rangle) \quad &= \quad \{v\} \cup L_\kappa(\kappa) \\ L_\kappa(\mathsf{arg}\langle - \; \rho \; \kappa\rangle) \quad &= \quad rng(\rho) \cup L_\kappa(\kappa) \\ L_\kappa(\mathsf{end}\langle - \; \kappa\rangle) \quad &= \quad L_\kappa(\kappa) \end{aligned}$$

Figure 9.20: Definitions for the *root values* $L(StA)$ of a step of the P-CEK$^q_{\text{NESL}}$ machine. This is a set of values, where labels act as roots into the store.

*Proof Outline:* To prove this we first generalize the statement to that of Lemma 9.2. There we consider the steps of P-CEK$^1_{\text{ArrL}}$ required to evaluate an expression in some general context and bound the reachable space during those steps by the space specified by the profiling semantics plus the control space at the beginning of the evaluation. The theorem then holds by specializing the lemma to start with an empty environment, store, and roots, and one active state. □

**Lemma 9.2** *If $e$ evaluates in the profiling semantics:*

$$\rho, \sigma, R \vdash e \overset{\text{ArrL}}{\longrightarrow} v, \sigma'; g, s,$$

*$e$ is a subexpression of an expression in the image of translation $T_{\text{ArrL}}[\![]\!]$, and for a step $i$ of the P-CEK$^1_{\text{ArrL}}$,*

- *the machine starts with a state or stub at the front of the active states stack that corresponds to this evaluation: $select(1, StA_i) = [(e, \rho, \kappa)], StA$, for some stack $StA$ and continuation $\kappa$,*

- *the semantics and machine can access the same locations: $L(StA_i) = R \cup \rho(FV(e))$, and*

- *these locations have the same values: $\forall_{l \in locs(L(StA_i), \sigma_i)} \sigma(l) = \sigma_i(l)$,*

*then*

- *on some future step $m \geq i$, the machine finishes this evaluation and calls $throw(v, \kappa)$,*
- *the maximum reachable space during this evaluation is bounded by the space for the original active states, plus a constant factor more than the space given by the profiling semantics:* $\max_{j=i}^{m} S_r(StA_j, \sigma_j) \leq S_A(StA_i) + k \cdot s$, *for some constant k.*

*Proof:* We prove this by structural induction on the language evaluation derivation and show a representative set of the cases. The remaining cases are similar.

**case VAR,** $e = x$**:** By the definition of the P-CEK$^q_{\text{ArrL}}$ machine, $throw(v, \kappa)$ is called on step $i$, so $m = i$. And by VAR, $s = S(R \cup \{\rho(x)\}, \sigma)$, so

$$\begin{array}{lll} \max_{j=i}^{m} S_r(StA_j, \sigma_j) & & \\ = & S_A(StA_i) + S(L(StA_i), \sigma_i) & \text{(Definition 6.11)} \\ = & S_A(StA_i) + S(L(StA_i), \sigma) & \text{(3rd assumption)} \\ = & S_A(StA_i) + s & \text{(2nd assumption)} \end{array}$$

The other base cases, CONST and ABSTR, are similar.

**case APP,** $e = e_1\ e_2$**:** Alternately inspecting the machine rules and using induction, we obtain the following results about the executions of the subexpressions $e_1$ and $e_2$ and on the appropriate function body $e_3$. The steps of the P-CEK$^1_{\text{ArrL}}$ corresponding to these three sub-evaluations are numbered $i_1$ to $m_1$, *etc.*, where $i_1 = i + 1$, $i_2 = m_1 + 1$, and $i_3 = m_2 + 2$, and step $m_2 + 1$ is the appropriate function call transition.

The active states at these important steps are as follows:

$$\begin{array}{rcl} StA_i & = & [(e_1\ e_2, \rho, \kappa)] +\!\!+ StA \\ StA_{i_1} & = & [(e_1, \rho, \mathsf{arg}\langle e_2\ \rho'\ \kappa \rangle)] +\!\!+ StA \\ StA_{i_2} & = & [(e_2, \rho, \mathsf{fun}\langle l\ \kappa \rangle)] +\!\!+ StA \\ StA_{m_2+1} & = & [(@\ l\ v_2, \cdot, \kappa)] +\!\!+ StA \\ StA_{i_3} & = & [(e_3, \rho'[x \mapsto l][y \mapsto v_2], \kappa)] +\!\!+ StA \end{array}$$

Furthermore, these three sub-evaluations result in the appropriate values:

- $l$ is the value of $e_1$, where $\sigma_{m_1}(l) = \mathbf{cl}(\rho',x,y,e')$ and $\rho' = restr(\rho, e')$;
- $v_2$ is the value of $e_2$; and
- $v$ is the result of the function body, and thus of the entire application.

We now look at the reachable space during the evaluation. First look at the steps not in the inductive sub-evaluations, *i.e.*, steps $i$ and $m_2 + 1$. Examining the definition of the P-CEK$^q_{\text{ArrL}}$ machine and using Definition 6.11, we have

$$\begin{array}{rcl} S_r(StA_i, \sigma_i) & = & S_r(StA_{i_1}, \sigma_{i_1}) \\ S_r(StA_{m_2+1}, \sigma_{m_2+1}) & \leq & S_r(StA_i, \sigma_i). \end{array}$$

So the reachable space in these steps is not greater than in the others.

Now we look at the reachable space in the inductive sub-evaluations. Using induction we have

$$\max_{j\in\{i_{j'},\ldots,m_{j'}\},j'\in\{1,2,3\}} S_r(StA_j,\sigma_j) \quad\leq\quad \max_{j'\in\{1,2,3\}} (S_A(StA_{i_{j'}})+k\cdot s_{j'}).$$

So we relate the control space at the beginning of these sub-evaluations, *i.e.*, $S_A(StA_{i_j})$, $j\in\{1,2,3\}$, to the control space of the starting step, $S_A(StA_i)$. For the first two sub-evaluations, $j\in\{1,2\}$, we see that

$$S_A(StA_{i_1}) = S_A(StA_{i_2}) = S_A(StA_i)+1.$$

For the space during the evaluation of the function body, $S_A(StA_{i_3})$, first observe that $|\rho'|+2\leq s_1$ by the definition of the store space since the closure with $\rho'$ must have been the result of a sub-derivation of $e_1$. Thus,

$$S_A(StA_{i_3})+k\cdot s_3 \quad\leq\quad S_A(StA_i)+k\cdot s$$

and the conclusion holds.

The APPC case, other than that for **addS**, is similar, but somewhat simpler, since it does not involve induction for the function body.

**case FORK,** $e = e_1\ e_2$ As in the APP case, we first use induction on the derivations of the two subexpressions $e_1$ and $e_2$. Then we use induction each of the $k$ forked branches of the body $e_3$. The steps of the P-CEK$^1_{\text{ArrL}}$ corresponding to these $k+2$ sub-evaluations are numbered $i_1$ to $m_1$, *etc.*, where $i_1 = i{+}1$, $i_2 = m_1{+}1$, $i_3 = m_2{+}2$, and $i_{k'+3} = m_{k'+2}{+}1$, for each branch $k'\in\{1,\ldots,k-1\}$.

The active states at these important steps are as follows:

$$\begin{array}{rcl}
StA_i & = & [(e_1\ e_2,\rho,\kappa)]{+\!\!+}StA\\
StA_{i_1} & = & [(e_1,\rho,\mathsf{arg}\langle e_2\ \rho'\ \kappa\rangle)]{+\!\!+}StA\\
StA_{i_2} & = & [(e_2,\rho,\mathsf{fun}\langle\mathbf{fork}\ \kappa\rangle)]{+\!\!+}StA\\
StA_{m_2+1} & = & [(@\ \mathbf{fork}\ l_2,\rho,\kappa)]{+\!\!+}StA\\
StA_{i_{k'+3}-1} & = & [\mathsf{@stub}\langle l\ k'\ (k-1)\ \mathsf{end}\langle l'\ \kappa\rangle\rangle]{+\!\!+}StA
\end{array}$$

for each branch $k'\in\{0,\ldots,k-1\}$. Note that this is just like the APP case for the first two uses of induction, except for the value of the function $e_1$:

- **fork** is the value of $e_1$, and
- $l_2$ is the value of $e_2$, where $\sigma_{m_2}(l_2) = \langle k,l'\rangle$, $\sigma_{m_2}(l') = \mathbf{cl}(\rho,\_,x,e_3)$ (the closure is guaranteed to use $\rho$ by definition of the translation $T_{\text{ArrL}}[\![]\!]$).

We now look at the reachable space during the evaluation. First look at the steps not in the inductive sub-evaluations, *i.e.*, steps $i$ and $i_{k'+3}-1$, for $k' \in \{0,\ldots,k-1\}$. Examining the definition of the P-CEK$^q_{\text{PAL}}$ machine and using Definition 6.11, we have

$$\begin{array}{rcl} S_r(StA_i, \sigma_i) & = & S_r(StA_{i_1}, \sigma_{i_1}) \\ S_r(StA_{i_{k'+3}-1}, \sigma_{i_{k'+3}-1}) & \leq & S_r(StA_{i_{k'+3}}, \sigma_{i_{k'+3}}) \end{array}$$

for each $k' \in \{0,\ldots,k-1\}$. So the reachable space in these steps is not greater than in the others.

Now we look at the reachable space in the inductive sub-evaluations. Using induction we have

$$\max_{j \in \{i_{j'},\ldots,m_{j'}\}, j' \in \{1,\ldots,k+2\}} S_r(StA_j, \sigma_j) \leq \max_{j' \in \{1,\ldots,k+2\}} (S_A(StA_{i_{j'}}) + k \cdot s_{j'}).$$

So we relate the control space at the beginning of these sub-evaluations, *i.e.*, $S_A(StA_{i_j})$, $j \in \{1, 2, k'+3\}$ for $k' \in \{0,\ldots,k-1\}$, to the control space of the starting step, $S_A(StA_i)$. Since all sub-evaluations start with the same environment,

$$\begin{array}{rcl} S_A(StA_{i_1}) & = & S_A(StA_i) + 1 \\ S_A(StA_{i_2}) & = & S_A(StA_i) + 1 \\ S_A(StA_{i_{k'+3}}) & = & S_A(StA_i) + 2 \end{array}$$

and the conclusion holds.

The APPC case for **addS** is similar.

□

### Parallel equivalence

Given the costs of serial execution in the abstract machine P-CEK$^1_{\text{PAL}}$, we are now concerned the costs of parallel execution, for P-CEK$^q_{\text{PAL}}$ with any $q$. Parallel execution can require more space because it can create many more simultaneous parallel threads (the active states stack can become much larger) and because it can have simultaneous access to many more locations in the store. We place bounds on how much extra space is needed.

As before, we show that the P-CEK$^q_{\text{ArrL}}$ executes a $q$-DFT traversal of the computation graph returned by the semantics, then use previous results on graph scheduling to bound the space. The following lemma and theorem provide bounds on the costs of the machine execution by showing that it corresponds to the specification of the computation graph

We also state that the profiling semantics and abstract machine compute the same value. The proofs concentrate on intensional aspects—we could add details of the extensional equivalence, as in the proof of serial equivalence.

**Lemma 9.3 (P-CEK$^q_{\mathrm{ArrL}}$ executes traversal)** *If e evaluates in the profiling semantics:*

$$\cdot, \cdot, \{\} \vdash e \xrightarrow{\mathrm{ArrL}} v, \sigma; g, s,$$

*and e is a subexpression of an expression in the image of translation $T_{\mathrm{ArrL}}[\![]\!]$, then it also evaluates in the abstract machine:*

$$\cdot, \cdot \vdash e \overset{\mathrm{ArrL},q}{\Longrightarrow} v, \sigma'; Q, \psi, s',$$

*such that the machine executes a q-traversal of the profiling semantics' graph g. I.e.,*

- *the selected states and visited nodes correspond at each step, and*
- *the active states (or in the case of stubs, the states represented by those stubs) and ready nodes correspond at each step.*

*Proof Outline:* We prove this by induction on the steps of the machine. We could fully formalize this as in Lemma 9.2.

For brevity, we refer to states being visited or ready, rather than corresponding to nodes which are visited or ready, respectively. Clearly the initial state is ready, as it corresponds to the source of $g$.

Inductively, we need to show that any states added to the active states stack are ready on the next step—the non-selected states left in the stack remain ready. By a case analysis on the expression of each of the selected states, we see that the computation substep generates states corresponding to the graph.

Constants, variables, and abstractions finish immediately, thus this state corresponds to the unit graph from the profiling semantics.

Applications generate one new state to start evaluating the function subexpression. This corresponds to the one child of the application node and is ready on the next step. Once it finishes, inductively, the machine has an active state which is ready and, when selected, starts the evaluation of the argument. Once the argument finishes, inductively, the machine generates a state for @ $v_1$ $v_2$ that is immediately ready, corresponding to the node before the function body. When the state is selected, the machine starts evaluating the function body, inductively (for a user-defined function, via $\delta$ (for a constant function other than **fork**), or as follows (for **fork**). For $v_1$ = **fork**, then the function body is the parallel branching of the argument's body. The state generated by the @ $v_1$ $v_2$ is a stub representing states corresponding to the the source node of each parallel branch. When selected, each inductively evaluates its branch. The last one to finish generates the state $(0, \cdot, \kappa)$ corresponding to the sink node of this parallel branching, which is immediately ready.

Pairs and conditionals similarly evaluate serially in the machine, as specified by the graph.
□

**Corollary 9.1** *If $e$ evaluates in the profiling semantics:*

$$\cdot,\cdot,\{\} \vdash e \xrightarrow{\text{ArrL}} v, \sigma; g, s,$$

*then it also evaluates in the abstract machine:*

$$\cdot,\cdot \vdash e \overset{\text{ArrL},q}{\Longrightarrow} v, \sigma'; Q, \psi, s',$$

*such that the number of states processed by the machine is the profiling semantics' work: $Q = W(g)$.*

*Proof:* This follows from the one-to-one correspondence of active states processed and nodes in the graph. □

**Theorem 9.3** (ArrL **executes** $q$-**DFT**) *If $e$ evaluates in the profiling semantics:*

$$\cdot,\cdot,\{\} \vdash e \xrightarrow{\text{ArrL}} v, \sigma; g, s,$$

*then it also evaluates in the abstract machine:*

$$\cdot,\cdot \vdash e \overset{\text{ArrL},q}{\Longrightarrow} v, \sigma'; Q, \psi, s',$$

*such that the machine executes a $q$-DFT of profiling semantics' graph $g$.*

*Proof:* This follows since the machine selects $\min(q, |StA|)$ nodes per step and since $g$ is series-parallel, together with Theorem 6.3 and Lemma 9.3. □

**Corollary 9.2** *If $e$ evaluates in the profiling semantics:*

$$\cdot,\cdot \vdash e \xrightarrow{\text{ArrL}} v, \sigma; g, s,$$

*then it also evaluates in the abstract machine:*

$$\cdot,\cdot \vdash e \overset{\text{ArrL},q}{\Longrightarrow} v, \sigma'; Q, \psi, s'$$

*such that the number of machine steps are bounded as a function of the profiling semantics' work and depth: $\psi \leq W(g)/q + D(g)$.*

*Proof:* This follows by Theorem 6.2. □

### Equivalence of space

Since the P-CEK$^q_{\mathrm{ArrL}}$ executes a traversal of the corresponding computation graph, we can use the machine to define the space costs of the graph nodes. Then using Theorem 6.4 we can bound the number of premature nodes on any given step of the P-CEK$^q_{\mathrm{ArrL}}$ and bound the memory used by these nodes, as Theorem 9.5 shows.

**Theorem 9.4** *Each step of a P-CEK$^q_{\mathrm{ArrL}}$ execution allocates at most $k$ space or deallocates at most $k$ space for each selected state, for some constant $k$.*

*Proof:* In the first substep, the cases calling *throw* create at most one new state, one new continuation, and one new intermediate state (which is deallocated later in the same step and can be ignored). It may also create a new restricted environment, which we assume to be of constant size, as discussed in Section 6.2. The other cases create at most one new state, two new environment bindings, and one new continuation. Note that the @ $l$ $v$ case need not create an entirely new environment, as environments can be shared, as discussed in Section 6.2. The substep may also allocate at most one new store binding.

For each selected state, the second substep deallocates any intermediate state created in the first substep. It may also allocate at most one new state or store binding. Note that which states create new states in this substep depends on the traversal.

Each selected state may also be credited with the deallocation of memory if this is the last state to reference it. This is a constant amount since each state refers to at most a constant amount of space. Note that we allow the crediting of a deallocation of a location even if it is still accessible, *e.g.*, in an environment or an an array. □

Since each step for a given selected state corresponds to a node (Lemma 9.3), each node allocates between $k$ and $-k$ space. By Theorem 9.2, the profiling semantics space is within a constant factor of the space complexity of the serial traversal. Thus as constant factors can be ignored, the profiling semantics space can be used in the context of Theorem 6.4 to provide a bound for the space of parallel execution.

**Theorem 9.5** (ArrL **parallel space)** *If*

- *program $e$ evaluates in the profiling semantics: $\cdot, \cdot, \{\} \vdash e \xrightarrow{\mathrm{ArrL}} v, \sigma; g, s$; and*
- *thus the program computes in the abstract machine: $\cdot, \cdot \vdash e \overset{\mathrm{ArrL},q}{\Longrightarrow} v, \sigma'; Q, \psi, s'$,*

*then the maximum reachable space in the abstract machine is bounded by the maximum reachable space of the profiling semantics plus a function of the parallelism: $s' \leq k(s + D(g)q)$.*

*Proof:* Since the P-CEK$^q_{\mathrm{ArrL}}$ machine executes a $q$-DFT of $g$, then by Theorem 6.1, on any step of the P-CEK$^q_{\mathrm{ArrL}}$ there can be at most $D(g)q$ nodes executed prematurely relative to the P-CEK$^1_{\mathrm{ArrL}}$. Since each state transition in step $i$ of a P-CEK$^q_{\mathrm{ArrL}}$ machine adds at most constant space to the next state of the machine, then the proof is easy. In particular since the maximum reachable space taken by any step of the P-CEK$^1_{\mathrm{ArrL}}$ is $kqs$, and on any step of the P-CEK$^q_{\mathrm{ArrL}}$ machine there are at most $D(g)q$ state transitions that were executed prematurely relative to some step of the P-CEK$^1_{\mathrm{ArrL}}$ machine, each of which allocated at most constant space, the total space is $k(s + D(g)q)$. □

## 9.5 Machine models

The implementation of this abstract machine is like that for the P-CEK$^q_{\mathrm{PAL}}$, *e.g.*, the active state stack is implemented with a SDGA. The two main differences are that the machine expand stubs when selecting states and handles the synchronization counters for the parallel branches of forks and scans.

**Theorem 9.6 (Cost of P-CEK$^q_{\mathrm{ArrL}}$ step)** *Each step of the P-CEK$^q_{\mathrm{ArrL}}$ machine can be processed on a $p$ processor machine in $O(q/p + TF(p))$ amortized time, w.h.p., and $O(q)$ maximum reachable space on the butterfly, hypercube, and PRAM machine models.*

*Proof:* The proof is like that for Theorem 7.1.

Each processor is responsible for at most $q/p$ of the current selected states, *i.e.*, processor $i$ is responsible for the states $[iq'/p, \ldots, (i+1)q'/p - 1]$, where $q' = \min(q, |StA|)$. We assume each processor knows its own processor number, so it can calculate a pointer to its section of the array.

The simulation of a step consists of the following phases, each of which we show can be executed with the given bounds:

1. Select the $q'$ states for this step, popping them from the stack and expanding the appropriate stub states.

   One way to efficiently expand the stub states is as follows:

   (a) Examine the first $\min(q + 1, |StA|)$ states, including stubs, in the active states stack. This represents at least $q'$ states since only the first stub can represent zero states.

   (b) Observe to how many states each expands.

   (c) Perform an add-scan to determine where the $q'$th stub occurs in the expansion, and expand the states up to that point.

   For practicality, the machine could cache $O(q)$ of expanded states that were not selected.

   This requires $O(q'/p)$ time and $O(q')$ space.

2. Locally evaluate the states using the $\overset{\mathrm{ArrL}}{\hookrightarrow}_{comp}$ transition, as in the P-CEK$^q_{\mathrm{PAL}}$.

   This is bounded by $O(q'/p + TS(p))$ time. Since at most one new state and one new location are created per selected state, this requires $O(q')$ space.

3. Locally evaluate the $\overset{\text{ArrL}}{\hookrightarrow}_{sync}$ transition.

   Here we must update the synchronization counters and merge the stores as if they were done sequentially. To update the counters we use the fetch-and-add operation. For the states ending a **fork** call, each adds $-1$ to the appropriate synchronization counter and fetches the new total number of elements left alive—the last state of each call creates a state for the dummy result and its continuation. For the states ending an **addS** call, each adds the appropriate data element to the appropriate running total. The last state of each call creates a state for the result and its continuation.

   Since each processor can have at most $q$ requests, this takes $O(q)$ time. The fetch-and-add can also be used for the transition on $\Sigma\langle\vec{i}, \vec{l'}, l'', i, \kappa\rangle$. For merging the stores the only operation that could conflict is a **store** instruction as part of implementing the **put** operation. However since the states have the same order as the processors, a priority concurrent write (with higher numbered processors given the higher priority) guarantees that rightmost value is written.

   Again, each processor accesses constant memory and allocates constant space. As in the P-CEK$^{q}_{\text{PAL}}$ machine, allocation can be eliminated by reusing the state that just resulted in this transition. If we avoid allocation, this phase requires $O(q'/p + TS(p))$ time, w.h.p., and $O(1)$ space.

4. Push the states created during this step onto the active state stack.

   This requires $O(q'/p + TF(p))$ amortized time, w.h.p., and $O(q')$ maximum space.

Adding the bounds for the three phases, we get the stated bounds for each of the machines. □

To account for memory latency in the butterfly and hypercube, and for the latency in the fetch-and-add operation for all three machines, we process $p \cdot TF(p)$ states on each step instead of just $p$ (*i.e.*, we use a P-CEK$^{p \cdot TF(p)}_{\text{ArrL}}$ machine).

**Corollary 9.3** *Each step of the P-CEK$^{p \cdot TF(p)}_{\text{ArrL}}$ machine can be simulated within $O(TF(p))$ amortized time on the $p$ processor butterfly, hypercube, and PRAM machine models, w.h.p.*

**Corollary 9.4** *If $e$ evaluates in the profiling semantics:*

$$\cdot, \cdot, \{\} \vdash e \xrightarrow{\text{ArrL}} v, \sigma; g, s,$$

*then the abstract machine evaluation*

$$\cdot, \cdot \vdash e \overset{\text{ArrL}, p \cdot TF(p)}{\Longrightarrow} v, \sigma'; Q, \psi, s'$$

*can be simulated within $O(W(g)/p + D(g)TF(p))$ amortized time and $O(s + p \cdot D(g)TF(p))$ maximum reachable space on the $p$ processor butterfly, hypercube, or PRAM machine models, w.h.p.*

*Proof:* Theorem 9.3 relates the graph $g$ to the P-CEK$^{q}_{\mathrm{ArrL}}$ computation, where $q = p \cdot TF(p)$. Theorem 6.2 bounds the number of steps of the graph traversal. There are $O(w/q + D(g))$ steps, and each step takes $O(TF(p))$ amortized time, w.h.p. Theorem 6.4 provides the space bound. □

# Chapter 10

# Algorithms and Comparing models

This chapter examines how we can use the PAL, PSL, and NESL models to analyze several parallel algorithms. Section 10.1 starts with a general discussion of how to analyze algorithms in these language models. As examples, Section 10.2 describes parallel versions of quicksort, mergesort, and Fast Fourier Transform (FFT) in each of the models. These illustrate some of the techniques necessary for programming efficient algorithms in the models. Then Section 10.3 defines a general relation of *cost-expressiveness* describing the relative power of language models, restates some of the previous results in terms of cost-expressiveness, and provides some additional general results.

## 10.1 Analyzing Algorithms

This section analyzes the work and depth bounds of several algorithms. Using the previous theorems on implementing these language models then obtains time bounds on the machine models. While not described here, space bounds could be obtained similarly.

Each profiling semantics defines a given computation's work and depth in terms of its computation graph. Recall that we have functions $W(g)$ and $D(g)$ defining the work and depth of a graph. For the PSL model, we also have the function $D'(g)$ defining the maximum depth. In each the algorithms discussed here, we are interested in the maximum depth for the PSL because we wish to wait until the entire result data structure is available.

We are generally interested in the work and depth, not of a single computation, but of a parameterized set of computations, *i.e.*, of an algorithm. In the PAL and NESL models, we use the profiling semantics to define recurrence equations that result in the algorithm's costs for any input. Recall that the computation graphs, and thus the work and depth costs, of these models are defined compositionally, as reflected in the recurrence equations. In the PSL model, the computation graphs are not defined compositionally. However, as Theorem 10.9 shows, the work of the PSL model is equivalent to that of the PAL model, so we can analyze it as such. To analyze the depth, we effectively "time-stamp" the data when it is created and see when the last data is created. More precisely, we consider the computation graph

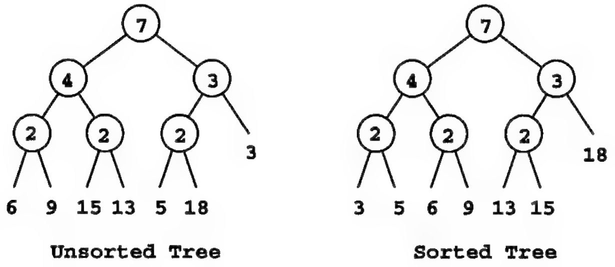


Figure 10.1: Representing sequences as balanced binary trees. The values are stored at the leaves, and each internal node stores the size of its subtree (the number of leaves below it).

where each node is labeled with its depth. Each computation node creates at most a single data object, so we treat the computation node's depth as the object's time-stamp.[1] For convenience, we idealize these time-stamps to ignore constant factor differences in depth, *e.g.*, identifying those at the same recursion depth in an algorithm when appropriate. Since the time-stamps correspond only to a subset of the computation nodes, we must also argue that the remaining computation does not dominate. That is simple if all of the algorithm is directed towards computing the single result, as in the examples here.

## 10.2 Specific Algorithms

This section describes the work and depth bounds of quicksort, mergesort, and FFT in each of the language models. We first note that any algorithm that examines all of its input, such as sorting or FFT, and represents its input as a list requires depth at least proportional to the number of elements in the input. In fact, a simple mergesort that makes its two recursive calls in parallel will match this lower bound for depth. To derive parallel algorithms that use time sublinear in the number of input elements (for other than the input and output) requires data structures other than lists. For the PAL and PSL models, we use balanced binary trees, as in Figure 10.1, where we assume that the ordering for sorted sequences in specified by a left-to-right traversal of the tree. For the NESL model, we could use trees, but sequences are clearly more efficient. This section shows how we can produce and analyze effective parallel versions of quicksort, mergesort, and FFT.

The complexities for the quicksort and mergesort algorithms are equivalent for the PAL and PSL. But the PSL requires asymptotically less depth than the PAL for the FFT algorithm. Furthermore, it is unclear whether a different FFT algorithm allows the PAL model to match the performance of the PSL.

For readability, the example code in this chapter uses pattern matching using the syntax of Standard ML. As used here, pattern matching can be encoded in any of the models with

[1] Note that an alternative way to define the PSL model is based on defining the depths of data objects rather than defining computation graphs [47], effectively including these time-stamps in the semantics.

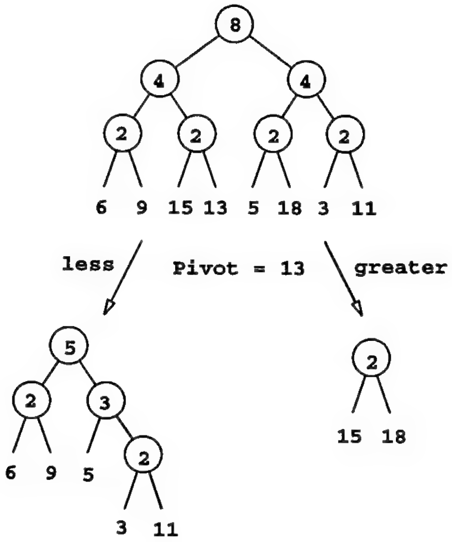


Figure 10.2: Quicksort pivoting. The algorithm chooses a pivot, here the median element, and then splits the tree into trees of lesser and greater elements.

only constant overheads.

### 10.2.1 Parallel Quicksort

The PAL and PSL code for our quicksort algorithm is given in Figure 10.3, and the code for NESL is given in Figure 10.4. As usual, the algorithm sorts by choosing an element to pivot on, selecting the lesser and greater elements, sorting those recursively, and appending the sorted elements (*cf.* Figure 10.2). The function `qsort_rec` returns a sorted tree, but in the PAL/PSL version, it will generally not be perfectly balanced, so the function `rebalance` rebalances it.

The function `select` selects all the elements of the tree matching the given predicate function. It calls itself recursively in parallel on both branches and append the results back together. Assuming the function `f` has constant work and depth, `select` on a tree of size $m$ and depth $d$ requires $O(m)$ work and $O(d)$ depth. We note that the tree returned by `select` is generally not going to be balanced, which is why we do not assume that $d = \log_2 m$. The `append` function simply puts its two arguments together in a tree node and therefore has constant work and depth.

We first present a general theorem that bounds work and depth in the PAL and PSL models for our quicksort in the expected case for any input tree, even if not balanced, and as a corollary give the bounds for balanced input.

**Theorem 10.1 (Quicksort in** PAL **and** PSL**)** *The quicksort algorithm* `quicksort` *shown in Figure 10.3, when applied to a tree with $m$ leaves and depth $d$, will execute in $O(m \log m)$*

```
datatype 'a Tree =
    Empty
  | Leaf of 'a
  | Node of int * 'a Tree * 'a Tree

fun elt (Leaf x) 1 = x
  | elt (Tree (_,xs,ys)) n = gt n (size xs) then elt ys (sub n (size xs)) else elt xs n

fun append Empty ys = ys
  | append xs Empty = xs
  | append xs ys = Tree (add (size xs) (size ys),xs,ys)

fun take _ 0 = Empty
  | take (Leaf x) 1 = Leaf x
  | take (Tree (_,xs,ys)) n =
    if gt n (size xs) then append xs (take ys (sub n (size xs))) else take xs n

fun drop xs 0 = xs
  | drop (Leaf x) 1 = Empty
  | drop (Tree (_,xs,ys)) n =
    if gt n (size ys) then drop xs (sub n (size ys)) else append xs (drop ys n)

fun select f Empty = Empty
  | select f (Leaf x) = if f x then Leaf x else Empty
  | select f (Tree (_,xs,ys)) = append (select f xs) (select f ys)

fun qsort_rec xs =
    if lt (size x) 2 then xs
    else let val pivot = elt xs (div2 (size xs))
             val lts = select (\ x. lt x pivot) xs
             val eqs = select (\ x. eq x pivot) xs
             val gts = select (\ x. gt x pivot) xs
         in append (qsort_rec lts) (append eqs (qsort_rec gts))

fun rebalance Empty = Empty
  | rebalance (Leaf x) = Leaf x
  | rebalance xs =
    let val half = div2 (size xs)
    in append (rebalance (take xs half) (rebalance (drop xs half))

fun quicksort xs = rebalance (qsort_rec xs)
```

Figure 10.3: PAL/PSL (tree-based) code for the parallel quicksort algorithm. This code uses syntactic extensions easily translatable into the syntax of the PAL and PSL.

```
fun qsort_rec xs =
    if lt (#xs,2) then xs
    else let val pivot = elt (xs,div2 (#xs))
             val lts = pack (xs,{lt (x,pivot) : x in xs})
             val eqs = pack (xs,{eq (x,pivot) : x in xs})
             val gts = pack (xs,{gt (x,pivot) : x in xs})
         in ++ (++ (qsort_rec lts,eqs), qsort_rec gts)

fun quicksort xs = qsort_rec xs
```

Figure 10.4: NESL quicksort algorithm. This codes uses syntactic extensions easily translatable into the syntax of NESL.

*work and $O(d \log m)$ depth in the* PAL *and* PSL *models, both expected case (i.e., averaged over all possible inputs of that depth and size).*

*Proof:* First consider qsort_rec. Note that since the pivots in quicksort will not perfectly split the data in general, some recursive paths will be longer than others. We call the longest path of recursive calls for qsort_rec on a particular input the *recursion depth* for that input. We note that the worst case recursion depth is $O(m)$ and that fewer than 1 of the $m$ possible inputs will lead to a recursion depth greater than $k \log m$, for some constant $k$ [102]. To determine the total computational depth of qsort_rec, we need to consider the computational depth along the longest path. We claim that this computational depth is at most $O(d)$ times the recursion depth since each node along the recursion tree will require at most $O(d)$ depth. This is because elt and select will run in $O(d)$ depth.[2] Since a fraction of only $1/m$ of the inputs will have a recursion depth greater than $O(\log m)$, and these cases will have recursion depth at most $O(m)$, the average (expected case) computation depth of qsort_rec is

$$\begin{aligned} D(d,m) &= O(d(\log m + \frac{1}{m}m)) \\ &= O(d \log m) \end{aligned}$$

To see that the work is expected to be $O(m \log m)$, we simply note that all steps do no more than a constant fraction more work than a list-based sequential implementation.

We now briefly consider the routine rebalance. We note that the depth of the tree returned by qsort_rec is at most a constant times the recursion depth. This is because the append operation that builds the returned tree simply joins the trees with a new node, not shuffling any elements. The function rebalance splits the tree along the path that separates the tree into two equal size pieces (or off by 1), recursively calls itself on the two parts, and

[2] Note that although select does not return balanced trees, it will never return a tree with depth greater than the original tree, which has depth $d$.

appends the results. Its costs are as follows:

$$\begin{aligned} W(m) &= W(\tfrac{m}{2}) + O(2(W_{\texttt{take}}(\tfrac{m}{2}) + W_{\texttt{drop}}(\tfrac{m}{2}))) \\ &= 2W(\tfrac{m}{2}) + O(m) \\ &= O(m \log m) \\ D(d,m) &= D(d', \tfrac{m}{2})) + O(\max(D_{\texttt{take}}(d,m), D_{\texttt{drop}}(d,m))), \text{ for some } d' < d \\ &= D(d', \tfrac{m}{2}) + O(d) \\ &\leq O(d \log m) \end{aligned}$$

For a tree of size $m$ and depth $d$, it requires $O(m \log m)$ work. It also requires at most $O(d \log m)$ depth, *e.g.*, when the tree is a chain, and at least $O(d)$ depth, when it is already balanced. □

**Corollary 10.1 (Quicksort in** PAL & PSL**)** *The quicksort algorithm* `quicksort` *shown in Figure 10.3, when applied to a balanced tree with m leaves, will execute in* $O(n \log m)$ *work and* $O(\log^2 m)$ *depth on the* PAL *and* PSL *models, both expected case.*

*Proof:* The depth of a balanced tree is $O(\log m)$. Thus, this gives an expected depth of $O(\log^2 m)$. □

**Theorem 10.2 (Quicksort in** NESL **[15])** *The quicksort algorithm* `quicksort` *shown in Figure 10.4, when applied to a tree with m leaves, will execute in* $O(m \log m)$ *work and* $O(\log m)$ *depth on the* NESL *model.*

### 10.2.2 Parallel Mergesort

We first consider the problem of merging two sorted trees. We use $m$ to refer to sum of the sizes of the two trees. We assume that each internal node contains the number and the maximum value of its descendants. This is clearly easy to generate in $O(m)$ work and $O(\log m)$ depth. The main component of the parallel algorithm is a routine `select_kth` which given two ordered trees $a$ and $b$, returns the $k^{\text{th}}$ smallest value from the combination of the two sequences (see Figure 10.5). It is implemented using a dual binary search in which we go down a branch from one of the two sequences on each step, using the maximal element at each node for navigation. Assuming the depths of the two trees are $d_a$ and $d_b$, the work and depth complexity of this routine is $O(d_a + d_b)$.

To merge two trees, we use `select_kth` to find their combined median element. We then select the elements less and greater, respectively, than the median for each tree with the functions `take_less` and `drop_less`. These can be implemented with $O(\log m)$ work and depth since the trees are sorted and balanced (it just requires going down a tree splitting along the way). Recursively merging the two trees of lesser elements and the two trees of greater elements gives us two sorted trees which are guaranteed to be the same size (or off by one) by construction. So, joining them under a new node produces a balanced sorted tree. As a whole, merging in this manner takes $O(m)$ work and $O(\log^2 m)$ depth since we recurse for the $\log_2 m$ depth of the trees.

**Theorem 10.3 (Mergesort in** PAL **&** PSL**)** *The mergesort algorithm* `mergesort` *shown in Figure 10.5, when applied to a balanced tree with $m$ leaves, will execute in $O(m \log m)$ work and $O(\log^3 m)$ depth on the* PAL *and* PSL *models.*

*Proof:* We can write the following recurrences for work and depth:

$$\begin{aligned} W(m) &= 2W(\tfrac{m}{2}) + W_{\mathtt{merge}}(m) \\ &= 2W(\tfrac{m}{2}) + O(m) \\ &= O(m \log m) \\ D(m) &= D(\tfrac{m}{2}) + D_{\mathtt{merge}}(m) \\ &= D(\tfrac{m}{2}) + O(\log^2 m) \\ &= O(\log^3 m) \end{aligned}$$

□

This version of mergesort is not as efficient as the quicksort previously described. However, if merging uses $O(m/\log m)$ splitters, rather than just the median, the depth complexities of merging and mergesort can each be improved by a factor of $O(\log m)$.

**Theorem 10.4 (More efficient mergesort in** PAL **and** PSL**)** *A mergesort algorithm on $m$ elements can execute in $O(m \log m)$ work and $O(\log^2 m)$ depth on the* PAL *and* PSL *models.*

*Proof:* The PAL case is shown by Blelloch and the author [16]. The PSL case then holds by Theorem 10.9. □

**Theorem 10.5 (Mergesort in** NESL **[17])** *A mergesort algorithm on $m$ elements can execute in $O(m \log m)$ work and $O(\log^2 m)$ depth on the* NESL *model.*

### 10.2.3 Fast Fourier Transform

This section presents the standard FFT algorithm, adapted to each of the models in the most straightforward manner, as shown in Figures 10.6 and 10.7. Each version assumes all inputs are of size $2^k$ for some integer $k$, as is usual for the FFT algorithm. This ensures that all trees are of size $2^j$ for some $j \leq k$ and are perfectly balanced. Therefore, we do not need to include `Empty` trees as in the previous sections. Unlike for quicksort and mergesort, here we show that this tree-based algorithm is more efficient in the PSL model than in the PAL model.

```
datatype 'a Tree =
    Empty
  | Leaf of 'a
  | Node of int * 'a * 'a Tree * 'a Tree

fun select_kth k (Leaf v1) (Leaf v2) =
    if gt v2 v1 then if eq k 0 then v1 else v0
    else if eq k 0 then v0 else v1
  | select_kth k (Leaf v1) (Node (n2,v2,l2,r2)) =
    if gt v2 v1 then if gt k n2
       then select_kth (sub k n2) (Leaf v1) r2
       else select_kth k (Leaf v1) l2
    else if gt n2 k
       then select_kth k (Leaf v1) l2
       else select_kth (sub k n2) (Leaf v1) r2
  | select_kth k (Node (n1,v1,l1,r1)) (Leaf v2) =
    select_kth k (Leaf v2) (Node (n1,v1,l1,r1)) =
  | select_kth k (Node (n1,v1,l1,r1)) (Node (n2,v2,l2,r2)) =
    if gt v2 v1 then if gt k (add n1 n2)
       then select_kth k (Node (n1,v1,l1,r1)) l2
       else select_kth (sub k n1) r1 (Node (n2,v2,l2,r2))
    else if gt k (add n1 n2)
       then select_kth k l1 (Node (n2,v2,l2,r2))
       else select_kth (sub k n1) (Node (n1,v1,l1,r1)) r2

fun merge (Leaf x) b = insert x b
  | merge a (Leaf y) = insert y a
  | merge a b =
    let val k = div2 (add (size a) (size b))
        val median = select_kth k a b
    in
        append (merge (take_less a median) (take_less b median))
               (merge (drop_less a median) (drop_less b median))

fun mergesort xs =
  if lt (size xs) 2 then xs
  else let val half = div2 (size xs)
       in merge (mergesort (take xs half)) (mergesort (drop xs half))
```

Figure 10.5: PAL/PSL (tree-based) code of the parallel mergesort algorithm. This code uses syntactic extensions easily translatable into the syntax of the PAL and PSL.

```
datatype 'a Tree =
    Leaf of 'a
  | Node of 'a * 'a Tree * 'a Tree

fun even_elts (Tree (Leaf x,Leaf y)) = Leaf y
  | even_elts (Tree (left,right)) = append (even_elts left) (even_elts right)

and odd_elts (Tree (Leaf x,Leaf y)) = Leaf x
  | odd_elts (Tree (left,right)) = append (odd_elts left) (odd_elts right)

fun map2 f (Leaf x) (Leaf y) = Leaf (f y w)
  | map2 f (Tree (xl,xr)) (Tree (yl,yr)) = append (map2 f xl yl) (map2 f xr yr)

fun fft (Leaf x) _ = Leaf x
  | fft xs ws =
    let rs1 = fft (even_elts xs) (even_elts ws)
        rs2 = fft (odd_elts xs) (even_elts ws)
    in map2 add (append rs1 rs1) (map2 mul (append rs2 rs2) ws)
```

Figure 10.6: PAL/PSL (tree-based) code for the parallel FFT algorithm, assuming the input size is a power of two. This code uses syntactic extensions easily translatable into the syntax of the PAL and PSL.

```
fun even_elts xs = pack ({x : x in xs},{even i : i in index (#xs)})

fun odd_elts xs = pack ({x : x in xs},{odd i : i in index (#xs)})

fun fft xs ws =
    if lt (#xs,2) then xs
    else let xys = {fft xs' (even_elts ws): xs' in [even_elts xs,odd_elts xs]}
         in {add x (mul y w): x in ++ (elt (xys,0),elt (xys,0))
                              y in ++ (elt (xys,1),elt (xys,1))
                              w in ws}
```

Figure 10.7: NESL (sequence-based) code for the parallel FFT algorithm.

**Theorem 10.6 (FFT in** PAL**)** *The FFT algorithm* `fft` *shown in Figure 10.6, when applied to balanced trees each with m leaves, will execute in $O(m \log m)$ work and $O(\log^2 m)$ depth on the* PAL *model.*

*Proof:* First, examine the helper functions. Each requires constant work per element and recurses on each half of its input. Thus they require $O(m')$ work and $O(\log m')$ depth on balanced trees of with $m'$ leaves.

The main function, `fft`, also recurses on each half of its input. But before recursing, it performs $O(m')$ work and $O(\log m')$ depth, assuming its input is balanced and has $m'$ leaves. Its input is balanced initially by assumption, and throughout the recursion by induction. Thus its costs are as follows:

$$\begin{aligned} W(m) &= 2W(\tfrac{m}{2}) + O(m) \\ &= O(m \log m) \\ D(m) &= 2D(\tfrac{m}{2}) + O(\log m) \\ &= O(\log^2 m) \end{aligned}$$

□

**Theorem 10.7 (FFT in** PSL**)** *The FFT algorithm* `fft` *shown in Figure 10.6, when applied to balanced trees each with m leaves, will execute in $O(m \log m)$ work and $O(\log m)$ depth on the* PSL *model.*

*Proof Outline:* Here we discuss only the depth bound, as the work bound holds by Theorems 10.6 and 10.9. We assume the whole input tree exists when starting the algorithm, so its time-stamps are all 1. Examining the time-stamps on the trees built during the algorithm, we show that the time-stamps on the output tree are $O(\log m)$.

We first examine time-stamps as the algorithm recurses on progressively smaller trees. Each call to `even_elts` and `odd_elts` creates a new tree by recursing down its input tree and finally selecting half of the leaves. They each build their result top-down, *i.e.*, the nodes at the top are created before those at the bottom. This is possible in the speculative model since each thread spawns child threads to build each subtree while this thread creates and returns a node. The time-stamp of each node in the result is the maximum of one more than the time when the algorithm recurses down to this level of the input tree and one more than the time-stamp of the needed data of the input tree. The top row of Figure 10.8 shows the time-stamps of trees during the recursive descent. This idealizes the amount of computation in these functions to a single step per node. It also idealizes computation by grouping function calls, such as the four calls to `even_elts` and `odd_elts`, into single steps and ignores additional overhead such as calls to `append`. As a result of the idealization, the time-stamps form a simple pattern down the tree. These idealized time-stamps are within a constant factor of the corresponding depths since each recursive call of the algorithm adds

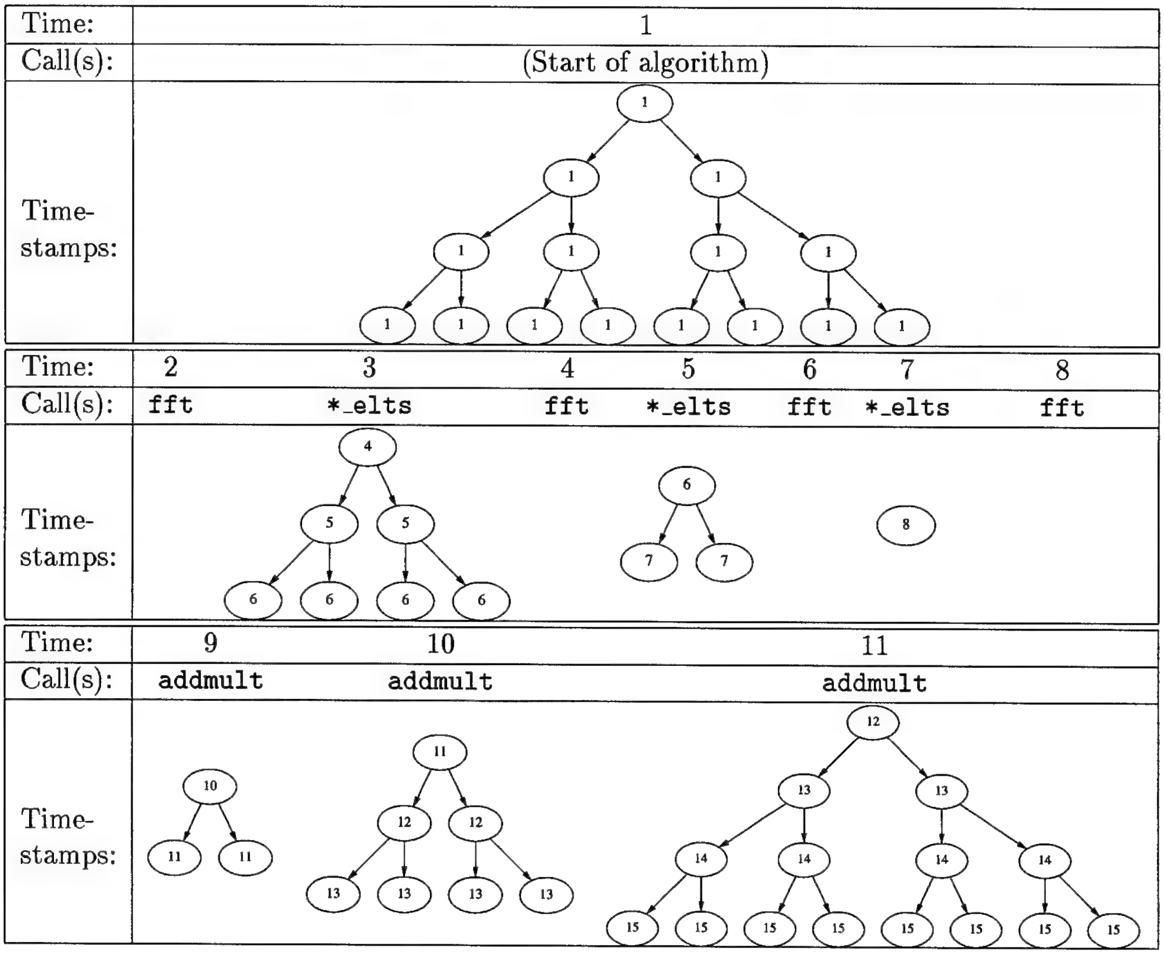


Figure 10.8: Time-stamps during PSL FFT algorithm. Tree nodes are labeled with their idealized time-stamps. The trees are the the result of the given function call made at the given idealized time.

only constant depth, *i.e.*, at most pattern matching the input, binding a constant number of variables, and calling a constant number of functions.

Next we examine time-stamps as the algorithm comes back up the recursion, using `addmult`. Each call to `addmult` creates a new tree by recursing down its input and finally performing the arithmetic. It also builds its result top-down. The bottom row of Figure 10.8 shows the time-stamps of tree during the recursive ascent, again idealizing computation as before.

The computation depth of the entire algorithm can now be broken into three components, each of which is $O(\log m)$:

- the depth of the recursive descent,
- the depth of the recursive ascent, and
- the delay for the threads to create the entire tree once the main thread is done.

Thus, the entire computation is of $O(\log m)$ depth. □

**Theorem 10.8 (FFT in** NESL**)** *The FFT algorithm of Figure 10.7 when applied to balanced trees each with m leaves will execute in $O(m \log m)$ work and $O(\log m)$ depth on the* NESL *model [15].*

## 10.3 Comparing models

Now that we have compared these three models on three specific algorithms, we make some more general statements about how the costs of these models relate. We then define and use a general relation of cost-expressiveness.

### 10.3.1 PAL and PSL

The previous comparisons, along with the definitions of computation graphs, suggest that the PSL model is more efficient than the PAL model. Here we show that this is the case for any given program, but possibly not in general. Note that we ignore the space of the PAL computation since we do not track the space of the PSL computation.

**Theorem 10.9 (Equivalence of** PAL **and** PSL**)** *If e evaluates in the* PAL *profiling semantics:*

$$\cdot, \cdot, \{\} \vdash e \xrightarrow{\text{PAL}} v, \sigma; g, -,$$

*then it also evaluates in the* PSL *profiling semantics:*

$$\cdot, \cdot \vdash e \xrightarrow{\text{PSL}} v', \sigma'; g'$$

*such that*

- *they compute with equal work:* $W(g) = W(g')$*,*
- *the* PSL *requires no more depth than the* PAL *to obtain a result:* $D(g) \geq D(g')$*, and*
- *the* PSL *requires no more depth than the* PAL *to terminate:* $D'(g) \geq D(g')$*.*

*Proof Outline:* This can be proved formally by generalizing the contexts to all environments and stores, using induction on the structure of $e$, which requires a case analysis on the form of $e$. In each case, we inspect the graphs formed by the appropriate pairs of semantics rules. For each kind of expression, the PSL graph has exactly as many nodes (work) and at most as many levels (depth) as the corresponding PAL graph.

For example, examine the graphs for the application $e_1\ e_2$ where $e_1$ evaluates to a closure, *i.e.*, those formed by the APP rule of each model. Induction shows that the theorem holds for the subcomputations $e_1$, $e_2$, and the body of the closure that $e_1$ evaluates to. Each APP rule creates a graph with the graphs of the subcomputations and exactly two new nodes. Thus the work of the graphs are equal. Each APP rule also adds edges to the result graph. The PSL adds three edges corresponding exactly to three of the PAL edges. The PSL may also add data edges via the VAR rule, but these impose no greater constraint that the fourth PAL edge, by the third conclusion of the $e_2$ induction. □

**Theorem 10.10** (PSL **sometimes faster than** PAL) *There exist algorithms which execute with asymptotically less depth in the* PSL *model than in the* PAL *model.*

*Proof:* Follows from Theorems 10.6 and 10.7. □

However, this does not prove that these programs cannot be rewritten in the PAL model to be as efficient as those in the PSL model. *I.e.*, this does not show the PAL model to be less efficient than the PSL in solving problems.

### 10.3.2 NESL and PAL

It should be clear that the NESL model is more time-efficient than the PAL model. First we show that NESL can simulate PAL with only constant cost overhead. That NESL is *strictly* more time-efficient then follows from the simple example of adding one to a collection of numbers, which requires only constant depth in the NESL model.

**Theorem 10.11** (PAL **and** NESL) *If $e$ evaluates in the* PAL *profiling semantics:*

$$\cdot, \cdot, \{\} \vdash e \overset{\mathrm{PAL}}{\longrightarrow} v, \sigma; g, s,$$

*then it can be simulated in the* NESL *profiling semantics:*

$$\cdot, \cdot \vdash T[\![e]\!] \overset{\mathrm{NESL}}{\longrightarrow} v', \sigma'; g', s'$$

*such that*

- *they compute within a constant factor of the same work:* $k \cdot W(g) = W(g')$, *and*
- NESL *requires no more than an extra constant factor extra depth:* $k \cdot D(g) \geq D(g')$,

*for some constant* $k$.

*Proof Outline:* To show this we need to give a translation $T[\![]\!]$ from the PAL model to the NESL model. This would be the identity, except for translating the PAL model's parallel application $e_1\ e_2$ in terms of the parallel NESL for-each:

$$\{x\ 0 : x \textbf{ in } [\lambda\_.e_1, \lambda\_.e_2]\}$$

where we use constant sequences for brevity. Clearly this introduces at most constant extra work and depth.

Note that this translation would not be typable in most type systems with homogeneous sequences (*e.g.*, as in existing NESL implementations). In the presence of types, a NESL simulation of the PAL model would need to encode the PAL expressions in some NESL type and then simulate the P-CEK$^q_{\text{PAL}}$ or a similar machine. □

The two models are equally space-efficient since they require equal space overhead (*cf.* Corollaries 7.2 and 9.4) and the PAL can encode any NESL sequence as a tree or list with only a constant factor space overhead for the pointers.

### 10.3.3 Simulation of traditional models

This section describes the simulation of a PRAM on the PAL. The simulation we use gives the same results for the EREW, CREW, and CRCW PRAM as well as for the multiprefix [100] and scan models [12]. The simulation is optimal in terms of work for all the PRAM variants. This is because it takes logarithmic work to simulate each random access into memory (this is the same as for pointer machines [7]). Since we don't know how to do better for the weaker models, we will base our results on the most powerful model, the CRCW PRAM with unit-time multiprefix sums (MP PRAM).

**Theorem 10.12 (PRAM simulation on** PAL) *A program that runs in time $t$ on a $p$ processor MP PRAM using $s$ space can be simulated on the* PAL *model with $O(p \log s)$ work and $O(t \log s \log p)$ depth.*

*Proof:* We will simulate a PRAM based on state transitions on the state $(C, S, P)$ where $C$ is the code, $S$ is the global state, and $P$ is per-processor state (*i.e.*, registers and program counter). Let $c = |C|$, $s = |S|$, and $p = |P|$, and assume that $c \leq s$ and $p \leq s$. For efficient access, the simulation stores $C$, $S$, and $P$ as balanced binary trees. Each state transition corresponds to a step of the PRAM, and the processors will be strictly synchronous. Register-to-register instructions can be implemented with $O(p)$ work and $O(\log p)$ depth, and concurrent reads with $O(p \log s)$ work and $O(\log s)$ depth. This just requires traversing the appropriate trees. The writes are the only interesting instruction to implement, and can be implemented by sorting the write requests from the processors by address and then recursively splitting the requests at each node of $S$ as we insert them. We can sort the $p$ requests in $O(p \log p)$ work and $O(\log^2 p)$ depth. We assume the sorted requests, which we call the write-tree $W$, start out balanced and are sorted from left to right in the tree. To implement a concurrent write or multiprefix, we combine nodes in the write-tree that have the same address. Since the addresses are sorted this can be done in $O(p)$ work and $O(\log p)$ depth.

We now consider the insertion of the sorted requests of a write-tree $W$ into state $S$ (*i.e.*, `modify(`$S$`,`$W$`)`). We assume that $S$ stores the addresses and associated values at the leaves, ordering the addresses from left-to-right, and that the internal nodes contain the value of the greatest address in the left branch. We assume all addresses in $W$ are also in $S$, and that each node of $W$ stores the minimum and maximum address of its descendants, so that we can access these in constant work and depth. To insert $W$ into $S$, we first check if $S$ is a single node, in which case $W$ must also be a single node, and we simply modify the value and return. Otherwise, we check if all the addresses in $W$ belong to just one of the branches of the $S$ tree. If so, we call `modify` recursively on that branch of $S$ with the same $W$ and put the result back together with the other branch of $S$ when the call returns. If not, we split $W$ based on the address stored at the root of $S$ and call `modify` in parallel on the two children of $S$ and the two split parts of $W$. This algorithm works since all addresses in the original write-tree will eventually find their way to the appropriate leaf of the $S$ tree and modify that leaf.

We now consider the total work and depth required. Splitting $W$ into two trees based on a key can be implemented in $O(\log p)$ work and depth by following down to the appropriate leaf, splitting along the way. Since $S$ is of depth $\log_2 s$, the total depth complexity is therefore bound by $O(\log p \log s)$. To prove the bounds on the work, we observe that it cannot take more than $O(p \log p)$ work to split the tree into $p$ pieces of size 1 since each split takes $O(\log p)$ work and there are $p - 1$ of them. This means the total work needed to split the original write-tree is bound by $O(p \log p)$. The only other work is the check at each node of the $S$ tree of whether we have to split or send all values down to one or the other branches. The maximum work done for these checks is $O(p \log s)$ since there can be at most $p$ separate chains (one per leaf of the write-tree) each which is at most as deep as the $S$ tree ($O(\log s)$) since $c \leq s$ and $p \leq s$. The total work is therefore $O(p(\log p + \log s)) = O(p \log s)$, since $p \leq s$. □

We now relate the PAL to another common parallel complexity class. NC is the class of problems solvable on a PRAM (any variant) in polylogarithmic time with a polynomial number of processors.

**Corollary 10.2** (PAL **relation to NC)** *Restricting the* PAL *model to those expressions that evaluate in polynomial work and polylogarithmic depth is equivalent to the NC complexity class.*

*Proof:* Any NC problem is solvable on a PRAM in polylogarithmic time with polynomial processors, by definition. By Theorem 10.12, this is solvable on the PAL within the desired bounds.

A PAL expression that requires polynomial work and polylogarithmic depth can be simulated on a PRAM in polylogarithmic time assuming we use as many processors as there is work, as seen from Corollary 7.2. □

These bounds also holds for the PSL model. If we knew how to sort faster on the PSL model than the PAL model, we could improve upon these bounds for the PSL. A similar simulation results in the same bounds for the NESL model.

### 10.3.4 Cost-expressiveness

We can generalize these comparisons by defining the notion of *cost-expressiveness*, which describes how efficiently models can simulate each other. *E.g.*, intuitively we would say that the NESL model is more time-expressive than the PAL model. This is based on extensional ideas of expressiveness, which are concerned with how languages compare in computational power.

Comparing two language models in isolation is meaningless because the abstract costs of the models have no *a priori* relation to each other. Rather, the comparison must occur relative to some machine model on which both language models are implemented. This gives a common base to compare costs. *E.g.*, an accurate restatement of the NESL/PAL relationship is that NESL is more time-expressive than and equally space-expressive as the PAL model, relative to any of the PRAM, hypercube, and butterfly. In contrast, the NESL and PAL models are equally work-, depth-, and space-expressive relative to the PAL model[3].

**Definition 10.1 (Cost-expressiveness)** *We now define when* model $A$ is at least as cost-expressive as model $B$, relative to implementation on model $C$, for implementation cost measure $c$, or $A \geq_{ce}^{C,c} B$. *For some implementation of* $B$ *in* $C$, *choose an implementation of* $A$ *in* $C$. *For all functions* $f$, *if some* $B$*-program computes* $f$ *with* $c$*-complexity* $O(c_B)$, *then some* $A$*-program computes* $f$ *with* $c$*-complexity* $O(c_A)$ *such that* $O(c_A) \leq O(c_B)$.

Note that we can choose a different implementation of $A$ for each implementation of $B$. While model $B$ may have an optimal implementation in model $C$, this is not always true. For example, while most NESL functions are faster in an implementation that uses pointer-based nested sequences, some like flattening and partitioning would be faster in an implementation that uses flattened nested sequences. The definition allows model $A$ different implementations for each of these for the model to be more cost-expressive than NESL.

Having a definition does not make general comparisons of cost-expressiveness easy. The following are some of the general statements that we can make:

- A non-Turing-equivalent model $A$ cannot be as or more cost-expressive than a Turing-equivalent model $B$, relative to any Turing-equivalent model, for any cost. This is

[3] Any model can be treated as a "machine" model!

| Relation | Follows from |
|---|---|
| PSLf $\geq_{ce}^{X,t}$ PAL | Th. 10.9, Cor. 7.2, Cor. 8.4 |
| PSLp $>_{ce}^{X,t}$ PSLf | Sec. 8.5.3 |
| NESL $>_{ce}^{X,t}$ PAL | Sec. 10.3.2, Cor. 7.2, Cor. 9.4 |
| NESL $=_{ce}^{X,s}$ PAL | Sec. 10.3.2 |
| PAL $>_{ce}^{X,t}$ NC | Cor. 10.2 |

Figure 10.9: Summary of PAL, PSL, and NESL cost-expressiveness. Here, $X$ is any of the butterfly, hypercube, and PRAM models of Section 7.1.

because model $B$ can compute more than $A$. However, $B$ is not necessarily more cost-expressive than $A$, because it may be less efficient for what $A$ can compute, thus they may be incomparable. This generalizes beyond Turing-equivalence to any idea of computability, *e.g.*, recursive enumerability.

- No model $A$ is more cost-expressive than model $B$ relative to model $B$, for any cost. *I.e.*, nothing can be simulated in $B$ any better than running native code.

- If model $A$ strictly extends model $B$ with additional features, then $A$ is at least as cost-expressive as $B$, relative to any other model and for any costs. The extra features in $A$ can be ignored for the comparison, but may give model $A$ an advantage.

Figure 10.9 summarizes some of our previous results, put in terms of cost-expressiveness.

# Part IV

# Conclusions

# Chapter 11

# Conclusions

We conclude with a summary of the contributions of this dissertation in Section 11.1 and a discussion of future extensions of this work in Section 11.2.

## 11.1 Summary of contributions

The primary conceptual contribution of this dissertation is the idea of provably efficient implementations. This extends previous work on providing provably correct implementations by also giving efficiency bounds on the implementation. Each provably efficient implementation consists of an abstract language model and a machine model, each with definitions of execution costs, and an implementation of the language model in the machine model and the cost mapping that induces.

This concept is useful for the language designers, implementors, and users (*i.e.*, programmers) alike. The designer is able to provide more complete specifications by describing intensional requirements. The implementor has a more complete specification, and this formal specification can be used to verify the extensional and intensional correctness of the implementation. The specification can also guide language development tools such as profilers and automatic complexity analyzers. It can also guide optimizations within the compiler. The programmer has an abstract summary of the costs of executing a program, together with mappings of these costs onto each target machine—the the programmer should not be expected to know the details of each compiler.

The dissertation describes three specific parallel models, and variants of these, based on fork-and-join parallelism, speculative parallelism, and data-parallelism (the PAL, PSL, and NESL models, respectively). While many language models could be described in the framework of provably efficient implementations, these models offer two core benefits. First, they are purely functional, and thus have relatively simple semantics which are suitable for use as intuitive examples. Second, they are parallel, with run-time costs of their implementations that are not readily apparent.

For each model, we provide a profiling semantics that defines its abstract costs. All three define the amount of computation and parallelism of the evaluation using computation graphs. The PAL and NESL models also define the amount of maximum reachable space during evaluation.

The definition of the PSL model is simpler than Roe's similar model, while more useful for describing parallelism. The PSL defines computation graphs, and thus the amount of parallelism available, unlike those of Roe [105, 106], Flanagan and Felleisen [37], and Moreau [84, 85], which only define the work of a computation. Similarly, the computation graphs in the NESL model are a more appropriate measure of parallelism than the maximum number of useful processors as defined by Zimmermann [128]. And this definition is formal, unlike that specified for NESL [14].

For each of these language models, we give a formal implementation onto the hypercube, butterfly, and PRAM machine models, and obtained the induced cost mapping. The speculative implementation is asymptotically faster than those used in practice, as it correctly parallelizes the suspension and reawakening of blocked threads. It is a much more detailed implementation that than of Moreau [84]. The data-parallel implementation is asymptotically more space-efficient than the current version of NESL.

We examine three examples—quicksort, mergesort, and Fast Fourier Transform—of how to program and analyze program costs in these models. We also define a general relation, called cost-expressiveness, of cost models based on the intensional aspects of the models and their implementations. We derive some classes of instances of the relation from its definition, and show additional instances relating the PAL, PSL, and NESL, based on the previous content of the dissertation.

In the future, we suggest that a definition of a language should include not only its extensional semantics, but also a profiling semantics and a provably efficient implementation.

## 11.2 Future work

We briefly discuss the practicality of the implementations and three types of extensions to this work: additional models, more detailed models, and automated use of such models.

### 11.2.1 Practicality of implementations

The dissertation has concentrated on asymptotic behavior at the cost of ignoring constant overheads. In particular, while we account for communication costs, we ignore the fact that communication is typically significantly slower than computation. Throughout the dissertation, we have mentioned possible improvements. Here we briefly discuss some additional pragmatic issues about the implementations and how they could be modified to reduce the constants.

The implementations aggressively create many threads to maximize parallelism and frequently synchronize all threads to guarantee load-balancing. Since most expressions are relatively simple, and the cost of thread management is high, creating a thread for each subexpression involves too much overhead (*e.g.*, [96]). Furthermore, each step performs little computation between each load-balancing. Thus the ratios of computation to both overhead and communication are relatively low.

One way to increase both ratios is grouping multiple sets of substeps between each load-balancing. *E.g.*, in the PAL implementation, a step could first evaluate each selected state two units of work, resulting in up to four new states, and then synchronize on all of these states. This reduces the number of load-balancing steps by half. In terms of the computation graph, this means that on each step the machine visits some set of ready nodes and then immediately visits the ready children of those nodes. This is the same basic idea as work examining heuristics for building large sequential blocks of code, *e.g.*, [57, 96]. As long as each step performs constant work per selected state, this does not effect our asymptotic time bounds, but reduces load-balancing costs and other overhead by a constant factor. Another improvement that is easy to incorporate is to avoid load-balancing whenever the number of active states is no greater than the number of processors.

We have also not addressed data locality. For example, one problem with the speculative model is that it accesses a remote threads to use those threads' final values. One improvement would be to cache those results in the local environment.

One way to reduce the space used is to introduce garbage collection of the various semantics objects. This can be done within the given work and space bounds.

Naturally, coding these implementations is necessary for more complete pragmatic comparisons. The implementations should be compared to those of languages such as Id, pH, and NESL, as appropriate.

### 11.2.2 Additional models

The framework of provably correct implementations could be used with any language model, and some specific models offer themselves as obvious next targets.

First, the speculative model could be adapted in two orthogonal ways:

- It could incorporate side-effects and/or continuations, in the style of Moreau [84, 85]. The only significant change required is adding *legitimacies* to ensure that side-effects and continuations happen in the same order as they would serially.[1] An important question is how much this serializes the implementation.

- It could be based on the more traditional implementation strategy of assigning threads to processors and letting them run until they finish or block. How can that strategy be adapted to execute within the bounds shown here?

Furthermore, it would be valuable to prove some results about the space usage of speculative computation and compare this to the other models. Is there a more space-efficient implementation of speculative computation that also adequately parallelizes?

Second, the data-parallel model could be adapted in three orthogonal ways:

[1] This requirement is simply an assumption made by Moreau and others.

- It could optimize some applications of **put** so that if its first argument value is not needed again, the application updates that sequence rather than creating a new one. When this case occurs, the application requires work proportional only to the number of updates rather than also to the size of the argument. It appears that a conservative reference counting can be implemented with only a constant factor overhead that is sufficiently accurate to detect some common cases when a sequence value is no longer needed. This results in asymptotically less work for some programs that repeatedly update sequences.

- It could be based on the implementation strategy of flattening nested data-parallelism, as in NESL. In the compilation to an ArrL-like language, nested uses of data-parallelism are compiled into single uses of data-parallelism on larger, flattened sequences. Its advantage is this increases the granularity of parallelism, often dramatically. The disadvantage is that it can asymptotically increase the size of sequences in the ArrL-like language, so that applications of constant functions can take asymptotically longer.

- Side-effects and other features could be added to model HPF, *etc.* As in the speculative model, the models would need to specify how any side-effects interact with parallelism.

The following examples are meant to hint at the range of additional properties and languages that are of interest:

- In languages with nested lexical scoping and first-class functions, how much space is required to store environments (bindings for variables) and how much time is required to add to or lookup from an environment?

- In functional languages, when is data copied and how much space do these copies take? This includes the question of whether tail recursion is used or not.

- In lazy languages, such as Haskell and Miranda, which expressions are executed is not readily apparent. Which are executed, how long will they take, in what order are they executed, and how much space does they use?

- In declarative languages such as Prolog, how long does searching for applicable rules, unifying terms, and backtracking take?

- In parallel languages, how long does communication take, and how does this affect the scheduling order of computations and thus other behaviors such as the allocation of memory?

With significantly different models, such as those based on object-oriented or declarative languages, we would use the same basic idea of defining an abstract semantics with cost information and relating this to machine models. Details such as the most appropriate form of an operational language semantics would likely need to change.

### 11.2.3 More detailed models

The models could also incorporate more detailed costs to accurately measure run-time costs, *i.e.*, worrying about constant factors. *I.e.*, provide a *microanalysis*, as opposed to our current *macroanalysis* [28]. At the simplest level, this could be distinguishing between the various costs here described as unit cost, *e.g.*, assigning some constant applications to be twice as expensive as others. More generally, this requires incorporating into the semantics anything considered relevant that affects the costs. At the extreme, this would include caches and processor pipelines. It could also include reflecting the distinctions made by compiler optimizations. While allowing more accurate definitions of given implementations, incorporating that level of detail negates the advantages of having an abstract language model.

Also, we could include more a formal treatment of garbage collection. This includes incorporating garbage collection formally into the implementations, proving the bounds outlined in Appendix B, and potentially improving upon these bounds.

### 11.2.4 Additional and more detailed comparisons of models

Within the given framework, additional comparisons of the models should be possible. In particular, we have only conjectured that the PSLf is *strictly* more time-expressive than the PAL. One way to show this would be proving that no PAL FFT algorithm requires only logarithmic depth. We have also not addressed how the PSL and NESL models compare. However, we conjecture that the NESL variant with the optimized **put** operation can simulate the PSL with $TF(p)$ work and depth overhead. The key is to have NESL simulate the PRAM using one large sequence to represent the PRAM memory. The optimization avoids any copying of this sequence.

Given more detailed models as previously described, we could also make more detailed comparisons.

### 11.2.5 Automated use of models

The original motivation for this work was to extend the work of automatic complexity analysis. Thus, a logical step would be to use these models as the core of automatic complexity analysis tools such as Metric, ACE, COMPLEXA, and $\Lambda_{\Upsilon}\Omega$ [124, 77, 35, 126], Kishon's profiling tool [66, 65], or compiler analyses.

# Bibliography

[1] Samson Abramsky and R. Sykes. Secd-m: A virtual machine for applicative programming. In Jean-Pierre Jouannaud, editor, *Proceedings 2nd International Conference on Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 81–98, 1985.

[2] Shail Aditya, Arvind, Jan-Willem Maessen, Lennart Augustsson, and Rishiyur S. Nikhil. Semantics of pH: A parallel dialect of Haskell. Technical Report Computation Structures Group Memo 377-1, Laboratory for Computer Science, Massachusetts Institute of Technology, June 1995.

[3] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[4] Boon S. Ang, Alejandro Caro, Stephem Glim, and Andrew Shaw. An introduction to the Id compiler. Technical Report Computation Structures Group Memo 328, Laboratory for Computer Science, Massachusetts Institute of Technology, May 1991.

[5] Andrew W. Appel. *Compiling with continuations.* Cambridge University Press, 1992.

[6] Henry G. Baker, Jr. and Carl Hewitt. The incremental garbage collection of processes. In *Proceedings of Symposium on AI and Programming Languages*, volume 12 of *SIGPLAN Notices*, pages 55–59, August 1977.

[7] Amir M. Ben-Amram and Zvi Galil. On pointers versus addresses. *Journal of the ACM*, 39(3):617–648, July 1992.

[8] Guy Blelloch, Phil Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 1–12, July 1995.

[9] Guy Blelloch and John Greiner. Parallelism in sequential functional languages. In *Proceedings 7th International Conference on Functional Programming Languages and Computer Architecture*, pages 226–237, June 1995.

[10] Guy Blelloch, Gary L. Miller, and Dafta Talmor. Developing a practical projection-based parallel Delaunay algorithm. In *Proceedings ACM Symposium on Computational Geometry*, May 1996.

[11] Guy Blelloch and Girija Narlikar. A comparison of two $n$-body algorithms. In *DIMACS Implementation Challenge Workshop*, October 1994.

[12] Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, C-38(11):1526–1538, November 1989.

[13] Guy E. Blelloch. *Vector Models for Data-Parallel Computing.* MIT Press, 1990.

[14] Guy E. Blelloch. NESL: A nested data-parallel language (version 3.1). Technical Report CMU-CS-95-170, Carnegie Mellon University, 1995.

[15] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, pages 85–97, March 1995.

[16] Guy E. Blelloch and John Greiner. A parallel complexity model for functional languages. Technical Report CMU-CS-94-196, Carnegie Mellon University, October 1994.

[17] Guy E. Blelloch and Jonathan C. Hardwick. Class notes: Programming parallel algorithms. Technical Report CMU-CS-93-115, Carnegie Mellon University, February 1993.

[18] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. In *Proceedings 25th ACM Symposium on Theory of Computing*, pages 362–371, May 1993.

[19] E. Börger and I. Durdanović. Correctness of compiling Occam to Transputer code. *The Computer Journal*, 39(1):52–92, 1996.

[20] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, 1974.

[21] Stephen Brookes and Shai Geva. Computational comonads and intensional semantics. Technical report, Carnegie Mellon University, 1991.

[22] F. W. Burton and D. J. Simpson. Space efficient execution of deterministic parallel programs. August 1996.

[23] F. Warren Burton. Guaranteeing good memory bounds for parallel programs. January 1996.

[24] Bettina Buth, Karl-Heinz Buth, Martin Fränzle, Burghard von Karger, Yassine Lakhneche, Hans Langmaack, and Markus Müller-Olm. Provably correct compiler development and implementation. In U. Kastens and P. Pfahler, editors, *Compiler Construction*, number 641 in Lecture Notes in Computer Science, pages 141–155. Springer-Verlag, 1992.

[25] Wentong Cai and David B. Skillicorn. Calculating recurrences using the Bird-Meertens formalism. *Parallel Processing Letters*, 5(2):179–190, June 1995.

[26] David Callahan and Burton Smith. A future-based parallel language for a general-purpose highly-parallel computer. In David Galernter, Alexander Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, Research Monographs in Parallel and Distributed Computing, chapter 6, pages 95–113. MIT Press, 1990.

[27] William Clinger and Jonathan Rees. Revised$^4$ report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July–September 1991.

[28] Jacques Cohen. Computer-assisted microanalysis of programs. *Communications of the ACM*, 25(10):724–733, October 1982.

[29] Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):31–53, 1986.

[30] James R. Driscoll, Neil Sarnak, Daniel D. K. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, February 1989.

[31] Marc Feeley. *An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors.* PhD thesis, Brandeis University, June 1993.

[32] Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17(1–3):35–75, December 1991.

[33] Matthias Felleisen and Daniel P. Friedman. A calculus for assignments in higher-order languages. In *Proceedings 13th ACM Symposium on Principles of Programming Languages*, pages 314–325, January 1987.

[34] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A report on the Sisal language project. *Journal of Parallel and Distributed Computing*, 10:349–366, 1990.

[35] Philippe Flajolet, Bruno Salvy, and Paul Zimmermann. Lambda-Upsilon-Omega: An assistant algorithms analayzer. *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, 357:201–212, June 1989.

[36] Philippe Flajolet, Bruno Salvy, and Paul Zimmermann. Automatic average-case analysis of algorithms. *Theoretical Computer Science*, 79(1):37–109, February 1991.

[37] Cormac Flanagan and Matthias Felleisen. The semantics of future and its use in program optimization. In *Proceedings 22nd ACM Symposium on Principles of Programming Languages*, pages 209–220, 1995.

[38] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings 10th ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[39] Daniel P. Friedman and D. S. Wise. The impact of applicative programming on multiprocessing. In *Proceedings International Conference on Parallel Processing*, pages 263–272, 1976.

[40] Daniel P. Friedman and D. S. Wise. Aspects of applicative programming for parallel processing. *IEEE Transactions on Computers*, 27(4):289–296, April 1978.

[41] Joseph Gil and Yossi Matias. Fast and efficient simulations among CRCW PRAMs. *Journal of Parallel and Distributed Computing*, 23(2):135–148, November 1994.

[42] Joseph Gil, Yossi Matias, and Uzi Vishkin. Towards a theory of nearly constant time parallel algorithms. In *IEEE Annual Symposium on Foundations of Computer Science*, pages 698–710, October 1991.

[43] T. Goldberg and U. Zwick. Optimal deterministic processor allocation. In *Proceedings 4th ACM-SIAM Symposium on Discrete Algorithms*, pages 220–228, January 1995.

[44] Michael T. Goodrich and S. Rao Kosaraju. Sorting on a parallel pointer machine with applications to set expression evaluation. In *Proceedings 30th IEEE Annual Symposium on Foundations of Computer Science*, pages 190–195, November 1989.

[45] Allan Gottlieb, B. D. Lubachevsky, and Larry Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems*, 5(2), April 1983.

[46] John Greiner. A comparison of parallel algorithms for connected components. In *Proceedings 6th ACM Symposium on Parallel Algorithms and Architectures*, pages 16–25, June 1994.

[47] John Greiner and Guy E. Blelloch. A provably time-efficient parallel implementation of full speculation. In *Proceedings 23rd ACM Symposium on Principles of Programming Languages*, pages 309–321, January 1996.

[48] Dale H. Grit and Rex L. Page. Deleting irrelevant tasks in an expression-oriented multiprocessor system. *ACM Transactions on Programming Languages and Systems*, 3(1):49–59, January 1981.

[49] Douglas J. Gurr. *Semantic Frameworks for Complexity.* PhD thesis, University of Edinburgh, January 1991.

[50] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[51] Robert H. Halstead, Jr. New ideas in Parallel Lisp: Language design, implementation, and programming tools. In T. Ito and R. H. Halstead, Jr., editors, *Parallel Lisp: Languages and Systems, US/Japan Workshop on Parallel Lisp*, number 441 in Lecture Notes in Computer Science, pages 2–51. Springer-Verlag, June 1989.

[52] Robert Hood. *The Efficient Implementation of Very-High-Level Programming Language Constructs.* PhD thesis, Cornell University, 1982.

[53] Paul Hudak and Steve Anderson. Pomset interpretations of parallel functional programs. In *Proceedings 3rd International Conference on Functional Programming Languages and Computer Architecture*, number 274 in Lecture Notes in Computer Science, pages 234–256. Springer-Verlag, September 1987.

[54] Paul Hudak and Robert M. Keller. Garbage collection and task deletion in distributed applicative processing systems. In *Proceedings ACM Conference on LISP and Functional Programming*, pages 168–178, 1982.

[55] Paul Hudak and Eric Mohr. Graphinators and the duality of SIMD and MIMD. In *Proceedings ACM Conference on LISP and Functional Programming*, pages 224–234, July 1988.

[56] Paul Hudak *et al.* Report on the functional programming language Haskell, version 1.2. *SIGPLAN Notices*, 27(5), May 1992.

[57] Lorenz Huelsbergen, James R. Larus, and Alexander Aiken. Using the run-time sizes of data structures to guide parallel-thread creation. In *Proceedings ACM Conference on LISP and Functional Programming*, pages 79–90, July 1994.

[58] Takayasu Ito and Manabu Matsui. A parallel lisp language PaiLisp and its kernal specification. In T. Ito and R. H. Halstead, Jr., editors, *Parallel Lisp: Languages and Systems, US/Japan Workshop on Parallel Lisp*, number 441 in Lecture Notes in Computer Science, pages 58–100. Springer-Verlag, June 1989.

[59] Joseph JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, Reading, MA, 1992.

[60] Neil D. Jones. Constant time factors *do* matter (extended abstract). In *Proceedings 25th ACM Symposium on Theory of Computing*, pages 602–611, May 1993.

[61] Mike Joy and Tom Axford. Parallel combinator reduction: Some performance bounds. Technical Report RR210, University of Warwick, 1992.

[62] A. R. Karlin and E. Upfal. Parallel hashing: an efficient implementation of shared memory. *Journal of the ACM*, 35:876–892, 1988.

[63] R. M. Karp and V. Ramachandran. Parallel algorithms for shared memory machines. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science — Volume A: Algorithms and Complexity*. MIT Press, Cambridge, MA, 1990.

[64] Richard Kennaway. A conflict between call-by-need computation and parallelism (extended abstract). In *Proceedings Conditional Term Rewriting Systems-94*, pages 247–261, February 1994.

[65] Amir Kishon. *Monitoring Semantics: Theory and Practice of Semantics-directed Execution Monitoring*. PhD thesis, Yale University, 1991.

[66] Amir Kishon, Paul Hudak, and Charles Consel. Monitoring semantics: A formal framework for specifying, implementing, and reasoning about execution monitors. In *Proceedings ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 338–352, June 1991.

[67] Jürgen Knopp. Improving the performance of parallel lisp by compile time analysis. In U. Kastnes and P. Pfahler, editors, *Compiler Construction*, volume 641 of *Lecture Notes in Computer Science*, pages 271–277. Springer-Verlag, 1992.

[68] Jürgen Knopp. Touching analysis: Avoiding runtime checking in future-based parallel languages. In Hesham El-Rewini, Ted Lewis, and Bruce D. Shriver, editors, *26th Proceedings Hawaii International Conference on System Sciences*, volume 2, pages 407–416. IEEE Computer Society Press, 1993.

[69] David A. Kranz, Jr. Robert H. Halstead, and Eric Mohr. Mul-T: A high-performance parallel lisp. In *Proceedings ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 81–90, June 1989.

[70] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6:308–320, 1964.

[71] Peter Lee and Uwe Pleban. A realistic compiler generator based on high-level semantics. In *Proceedings 14th ACM Symposium on Principles of Programming Languages*, pages 284–299, 1987.

[72] F. T. Leighton, B. M. Maggs, A. G. Ranade, and S. B. Rao. Randomized routing and sorting on fixed-connection networks. *Journal of Algorithms*, 17(1):157–205, July 1994.

[73] Yossi Matias and Uzi Vishkin. On parallel hashing and integer sorting. *Journal of Algorithms*, 12(4):573–606, December 1991.

[74] Yossi Matias and Uzi V'shkin. A note on reducing parallel model simulations to integer sorting. In *Proceedings 9th International Parallel Processing Symposium*, pages 208–212, April 1995.

[75] Kurt Mehlhorn and Uzi Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memory. *Acta Informatica*, 21:339–374, 1984.

[76] Daniel Le Métayer. Mechanical analysis of program complexity. In *Proceedings SIGPLAN Symposium on Language Issues in Programming Environments*, 1985.

[77] Daniel Le Métayer. Ace: An automatic complexity evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, April 1988.

[78] Daniel Le Métayer. Analysis of functional programs by program transformation. In J.-P. Banâtre, S. B. Jones, and D. Le Métayer, editors, *Prospects for Functional Programming in Software Engineering*, volume 1 of *Research Reports, ESPRIT, Project 302*, chapter 5, pages 87–120. Springer-Verlag, 1991.

[79] James S. Miller. *MultiScheme: A Parallel Processing System Based on MIT Scheme.* PhD thesis, Massachusetts Institute of Technology, September 1987.

[80] Peter H. Mills, Lars S. Nyland, Jan F. Prins, John H. Reif, and Robert A. Wagner. Prototyping parallel and distributed programs in Proteus. Technical Report UNC-CH TR90-041, Computer Science Department, University of North Carolina, 1990.

[81] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1990.

[82] John C. Mitchell. On abstraction and the expressive power of programming languages. In *Proceedings Theoretical Aspects of Computer Software*, pages 290–310, September 1991.

[83] Luc Moreau. The PCKS-machine. an abstract machine for sound evaluation of parallel functional programs with first-class continuations. In *European Symposium on Programming*, number 788 in Lecture Notes in Computer Science, pages 424–438. Springer-Verlag, April 1994.

[84] Luc Moreau. The semantics of Scheme with future. Technical Report M95/7, Department of Electronics and Computer Science, University of Southampton, 1995.

[85] Luc Moreau. The semantics of Scheme with future. In *Proceedings 1st ACM SIGPLAN International Conference on Functional Programming*, pages 146–156, May 1996.

[86] Rishiyur S. Nikhil. The parallel programming language Id and its compilation for parallel machines. Technical Report Computation Structures Group Memo 313, Massachusetts Institute of Technology, July 1990.

[87] Rishiyur S. Nikhil. Id version 90.1 reference manual. Technical Report Computation Structures Group Memo 284-1, Laboratory for Computer Science, Massachusetts Institute of Technology, July 1991.

[88] Rishiyur S. Nikhil, Arvind, James Hicks, Shail Aditya, Lennart Augustsson, Jan-Willem Maessen, and Yuli Zhou. pH language reference manual, version 1.0—preliminary. Technical Report Computation Structures Group Memo 369, Laboratory for Computer Science, Massachusetts Institute of Technology, January 1995.

[89] Chris Okasaki. Simple and efficient purely functional queues and dequeues. *Journal of Functional Programming*, 5(4):583–592, October 1995.

[90] Randy B. Osborne. *Speculative Computation in Multilisp.* PhD thesis, Massachusetts Institute of Technology, December 1989.

[91] Robert Paige. Real-time simulation of a set machine on a RAM. In W. Koczkodaj, editor, *Proceedings International Conference on Computing and Information*, volume 2, pages 68–73, 1989.

[92] Michel Parigot. Programming with proofs: A second order type theory. In H. Ganzinger, editor, *Proceedings 2nd European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 145–159. Springer-Verlag, 1988.

[93] Andrew S. Partridge. *Speculative Evaluation in Parallel Implementations of Lazy Functional Languages.* PhD thesis, Department of Computer Science, University of Tasmania, 1991.

[94] Andrew S. Partridge and Anthony H. Dekker. Speculative parallelism in a distributed graph reduction machine. In *Proceedings Hawaii International Conference on System Sciences*, volume 2, pages 771–779, 1989.

[95] Lawrence C. Paulson. A semantics-directed compiler generator. In *Proceedings 9th ACM Symposium on Principles of Programming Languages*, pages 224–239, January 1982.

[96] Simon L Peyton Jones. Parallel implementations of functional programming languages. *The Computer Journal*, 32(2):175–186, 1989.

[97] Nicholas Pippenger. Pure versus impure lisp. In *Proceedings 23rd ACM Symposium on Principles of Programming Languages*, pages 104–109, January 1996.

[98] Uwe F. Pleban and Peter Lee. An automatically generated, realistic compiler for an imperative programming language. In *Proceedings ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 23 of *SIGPLAN Notices*, pages 222–227, June 1988.

[99] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1, August 1974.

[100] Abhiram G. Ranade. *Fluent Parallel Computation.* PhD thesis, Yale University, New Haven, CT, 1989.

[101] Abhiram G. Ranade. How to emulate shared memory. *Journal of Computer and System Sciences*, 42(3):307–326, June 1991.

[102] Rudiger Reischuk. Probabilistic parallel algorithms for sorting and selection. *SIAM Journal of Computing*, 14(2):396–409, 1985.

[103] Oege de Moor Richard Bird, Geraint Jones. A lazy pure language versus impure lisp. Post in `comp.lang.functional` newsgroup, May 1996.

[104] J. W. Riely, J. Prins, and S. P. Iyer. Provably correct vectorization of nested-parallel programs. In *Proceedings Programming Models for Massively Parallel Computers*, pages 213–222. IEEE Computer Society Press, October 1995.

[105] Paul Roe. Calculating lenient programs' performance. In Simon L Peyton Jones, Graham Hutton, and Carsten Kehler Holst, editors, *Proceedings Functional Programming, Glasgow 1990*, Workshops in computing, pages 227–236. Springer-Verlag, August 1990.

[106] Paul Roe. *Parallel Programming using Functional Languages.* PhD thesis, Department of Computing Science, University of Glasgow, February 1991.

[107] John R. Rose and Guy L. Steele Jr. C*: An extended C language for data parallel programming. In *Proceedings 2nd International Conference on Supercomputing*, volume 2, pages 2–16, May 1987.

[108] Mads Rosendahl. Automatic complexity analysis. In *Proceedings 4th International Conference on Functional Programming Languages and Computer Architecture.* Springer-Verlag, September 1989.

[109] David Sands. Complexity analysis for a lazy higher-order language. In *Proceedings Functional Programming, Glasgow 1989*, Workshops in Computing Series. Springer-Verlag, 1989.

[110] David Sands. *Calculi for Time Analysis of Functional Programs.* PhD thesis, University of London, Imperial College, September 1990.

[111] David Sands. Time analysis, cost equivalence and program refinement. In *Proceedings 11th Conference on Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science. Springer-Verlag, December 1991.

[112] Patrick M. Sansom. *Execution Profiling for Non-strict Functional Languages.* PhD thesis, Department of Computing Science, University of Glasgow, 1994.

[113] Patrick M. Sansom and Simon L Peyton Jones. Time and space profiling for non-strict, higher-order functional languages. In *ACM Symposium on Principles of Programming Languages*, 1995.

[114] Helmut Seidl and Reinhard Wilhelm. Probabilistic load balancing for parallel graph reduction. In *Proceedings TENCON '89, 4th IEEE Region 10 International Conference*, pages 879–884, November 1989.

[115] Jon Shultis. On the complexity of higher-order programs. Technical Report CU-CS-288-85, University of Colorado, Boulder, January 1985.

[116] David B. Skillicorn. The Bird-Meertens formalism as a parallel model. In *Proceedings Software for Parallel Computation*, June 1992.

[117] David B. Skillicorn and W. Cai. A cost calculus for parallel functional programming. *Journal of Parallel and Distributed Computing*, 28(1):65–83, July 1995.

[118] Dan Suciu and Val Tannen. Efficient compilation of high-level data parallel algorithms. In *Proceedings 6th ACM Symposium on Parallel Algorithms and Architectures*, pages 57–66, June 1994.

[119] Carolyn Talcott. *Rum*: An intensional theory of function and control abstractions. In *Proceedings Workshop on Foundations of Logic and Functional Programming.* Springer-Verlag, 1986.

[120] Kenneth R. Traub. *Sequential Implementation of Lenient Programming Languages.* PhD thesis, Massachusetts Institute of Technology, October 1988.

[121] Guy Tremblay and G. R. Gao. The impact of laziness on parallelism and the limits of strictness analysis. In A. P. Wim Bohm and John T. Feo, editors, *Proceedings High Performance Functional Computing*, pages 119–133, April 1995.

[122] L. G. Valiant. A scheme for fast parallel communication. *SIAM Journal of Computing*, 11(2):350–361, May 1982.

[123] L. G. Valiant. *General Purpose Parallel Architectures*, volume A, chapter 18, pages 943–972. Elsevier Science Publishers, 1990.

[124] Ben Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9), September 1975.

[125] C. K. Yuen, M. D. Feng, and J. J. Yee. Speculative parallelism in BaLinda Lisp. Technical Report TR31/92, Department of Information Systems and Computer Science, National University of Singapore, November 1992.

[126] Paul Zimmermann and Wolf Zimmermann. The automatic complexity analysis of divide-and-conquer algorithms. Technical Report 1149, Institut National de Récherche en Informatique et en Automatique, Rocquencourt, December 1989.

[127] Paul Zimmermann and Wolf Zimmermann. The automatic complexity analysis of divide-and-conquer algorithms. *Computer and Information Sciences VI*, 1:395–404, November 1991.

[128] Wolf Zimmermann. Automatic worst case complexity analysis of parallel programs. Technical Report TR-90-066, International Computer Science Institute, December 1990.

[129] Wolf Zimmermann. The automatic worst case analysis of parallel programs: Simple parallel sorting and algorithms on graphs. Technical Report TR-91-045, International Computer Science Institute, August 1991.

[130] Wolf Zimmermann. Complexity issues in the design of functional languages with explicit parallelism. In *Proceedings International Conference on Computer Languages*, pages 34–43, April 1992.

# Appendix A

# Glossary

This appendix briefly describes several array operations which are used in the machine model implementations and the data-parallel language. See Figure 8.1 for some time bounds on implementing the fetch-and-add operation. See Figure 7.4 for time bounds on the other operations.

**fetch-and-add** In the fetch-and-add [45] (or multiprefix [100]) operation, each processor has an address and an integer value $i$.

Consider each set of processors using a given address. Each processor receives the sum of the initial contents at the address and the values of the processors writing to the address prior to this processor. The final value stored at the address is its initial contents plus the sum of the appropriate processors' values. Here, we assume that the fetch-and-add is *stable*—that lower-numbered processors access their addresses first. For example, a fetch-and-add of an array of addresses $[1,0,2,1,1,2]$ and array of values $[3,8,4,12,5,7]$, where the addresses' initial contents are zeroes, results in the new array $[0,0,0,3,15,4]$ and the addresses' contents being 8, 20, and 11.

This can also be described as having each processor, in parallel, atomically fetch its address' contents and increment it by $i$. The stable fetch-and-add operation can be implemented in a butterfly or hypercube network by combining requests as they go through the network [100], and on a PRAM by various other techniques [73, 41]. If each processor has at most $m$ data elements to combine, all data can be processed in $O(m + \log p)$ time on the butterfly and hypercube, and in $O(m + \log p / \log\log p)$ time on the CRCW PRAM, each with high probability. We parameterize these bounds to say that a fetch-and-add on $m \cdot p$ elements requires $O(m + TF(p))$ time. For values of $TF(p)$, see Figure 8.1.

In the degenerate case where all processors use the same address, this implements a scan (prefix sum) and reduce on the integer data values. More generally, a fetch-and-op is defined and implemented in the same way for any associative binary operation.

**scan** A scan operation (or prefix sum) combines each prefix of an array with an associative binary operation. For example, the additive scan of $[2,1,3]$ is $[0,2,3,6]$.

This is equivalent to a fetch-and-add with all processors using the same address, and the address' contents initially zero.

If each processor has at most $m$ data elements to combine, all data can be processed in $O(m + \log p)$ time on the butterfly and hypercube, and in $O(m + \log p / \log\log p)$ time on the CRCW PRAM. We parameterize these bounds to say that a scan on $m \cdot p$ elements requires $O(m + TS(p))$ time. For values of $TS(p)$, see Figure 7.4.

**reduce** A reduce operation combines all elements of an array. For example, the additive reduction of $[2,1,3]$ is 6.

**index** An index operation takes an integer $i$ and creates an array $[0, \ldots, i-1]$ of that length.

A segmented index of the array $[2,1,3]$ performs an index operation on each element 2, 1, and 3, and combines the results into an array $[[0,1],[0],[0,1,2]]$.

**distribute** A distribute operation takes a value $v$ and an integer $i$ and creates an array $[v, \ldots, v]$ of $i$ copies of the value.

A segmented distribute of the values $[v_0, v_1, v_2]$ and the array $[2,1,3]$ distributes each of the values as follows: $[[v_0, v_0], [v_1], [v_2, v_2, v_2]]$.

**pack** A pack operation takes an array of values and an array of booleans of the same length and returns an array containing the values whose corresponding flag is **true**. The result's contents are in the same order as in the original array.

**put** A put operation takes an array of values $\vec{v'}$ and an array of pairs, each with an index $j_i$ and value $v_i$. The operation creates a an array like the original one except that it contains the values $\vec{v}$ at the corresponding indices. Thus, each index $j_i$ must be in the range $0, \ldots, |\vec{v'}|$. If a given index occurs multiple times, we specify that the last corresponding value is used. So, for example. the put of $[2,1,3]$ and $[(1,9),(0,6),(1,7)]$ is $[6,7,3]$.

Scans, reduces, indices, and distributes can all be implemented with a constant number of fetch-and-op operations.

# Appendix B

# Simple Parallel Garbage Collection

This appendix describes a parallel version of garbage collection (GC) and its effects on the machine complexity bounds for the PAL and NESL models. It maintains the space bounds, but adds work proportional to the number of collections (*gc*) during the evaluation. This algorithm is only outlined, and its details should be further explored.

The machine has a block of memory available for use (the *heap*) and consists of allocated memory and unallocated memory. The heap is initially of some constant size. During an allocation, if there is insufficient unallocated space in the heap, the machine increases the heap size and garbage collects the current allocated data. The heap doubles in size during each GC to twice the size sufficient for the allocation, like a SDGA (*cf.* Section 7.2. The heap never shrinks, although the allocated space within it may.

We store the continuation stacks within the heap[1], and assume that each processor has a constant number of registers. This ensures that there are at most $O(p)$ root pointers into the heap—the profiling semantics' roots are all stored in these continuations. Space for the continuation stack is already accounted by the added space constants in the profiling semantic rules, so placing it in the heap does not alter the previous space bounds.

In *stop-and-copy* GC, the heap consists of two sub-blocks: the allocated data and the unallocated space. In each of languages examined here, the heap and roots form a forest of data. On each allocation, if there is enough space available in the heap, that space is allocated. Otherwise, the machine increases the size of the heap and copies the surviving data into the new empty part of the heap, as illustrated in Figure B.1. The old data is copied by a $p$-DFT of this forest.

Each GC clearly requires at least $\lceil m_1/p \rceil$ time to examine and potentially copy $m_1$ data. Over all GCs, this is bounded by $O(w/p)$ time, where $w$ is the work of the entire program, since the heap doubles in size on each GC. In the PAL and NESL models, the data forest is of at most $d$ depth, where $d$ is the computation depth of the entire program. Thus copying can require a constant factor times $d \cdot TS(p)$ (for PAL model) or $d \cdot TF(p)$ (for NESL model) time. So, GC requires at most

[1] *E.g.*, as in the implementation of Standard ML of New Jersey.

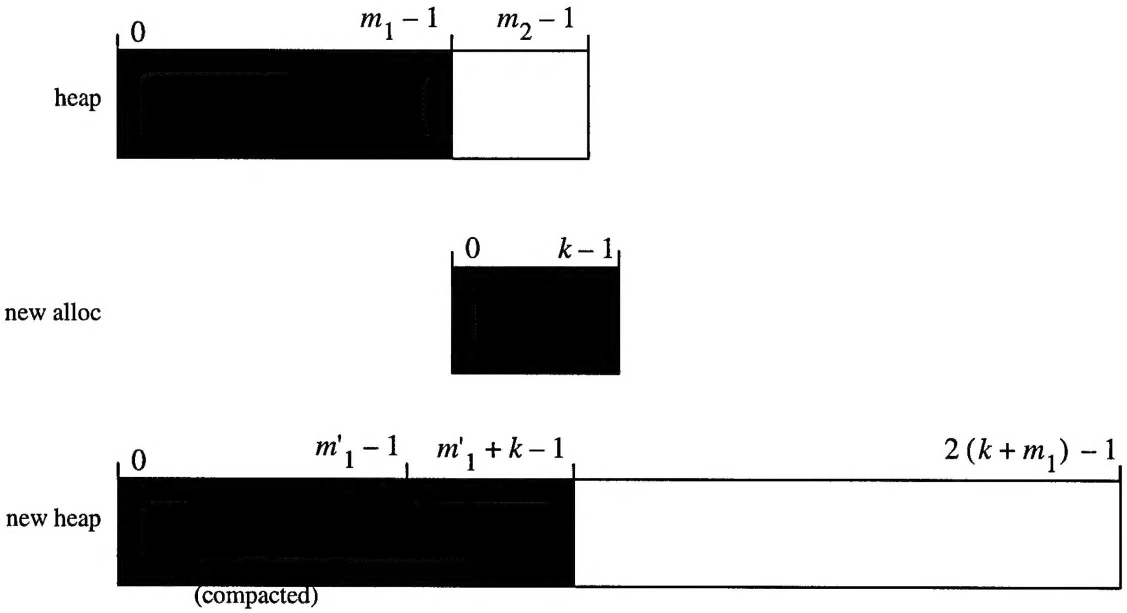


Figure B.1: Basic structure of stop-and-copy garbage collection. The allocated space is marked with lines, while unallocated space is blank.

$$\begin{array}{ll} O(w/p + gc \cdot d \cdot TS(p)) & \text{(for PAL model)} \\ O(w/p + gc \cdot d \cdot TF(p)) & \text{(for NESL model)} \end{array}$$

time, which dominates the overall evaluation time.

Since the heap doubles in size per GC and since each model ensures that we spend $m$ work to allocate $m$ space, there may be at most $gc = O(\log W(g))$ GCs. Thus, GC and the overall computation requires at most

$$\begin{array}{ll} O(w/p + \log w \cdot d \cdot TS(p)) & \text{(for PAL model)} \\ O(w/p + \log w \cdot d \cdot TF(p)) & \text{(for NESL model)} \end{array}$$

time.

Since the heap doubles in size on each GC, then after the first GC, the heap size is never more than twice the maximum reachable space. *I.e.*, with stop-and-copy GC, the total space complexity is asymptotically the same as the maximum reachable space complexity without GC for the PAL and NESL models.